Software Transactional Memory

CSCI 5828: Foundations of Software Engineering
Lecture 19 — 03/20/2012

Goals

- Review material in Chapter 6 of our concurrency textbook
- Introduce Software Transaction Memory
  - Separation of Identity and State to enable this approach
  - Discuss the lock free programming of concurrent systems it enables
- Review several examples both in Clojure and Java

Software Transactional Memory

- The **problems** associated **with shared mutability** in concurrent software systems have **led computer scientists to invent alternatives**
- One such approach is known as the **software transactional memory**
  - this approach to concurrency was popularized by its inclusion into the runtime of the Clojure programming language
  - frameworks which implement STM are available for other programming languages, including Java and Scala
- STM provides a means for explicitly **keeping track of mutable state** and **ensuring** that **changes to that state are protected and visible** to all threads

When is STM useful?

- STM is best used in those applications in which the **access patterns** to **shared mutable state** follow this pattern
  - **frequent reads** (by multiple threads)
  - **very infrequent write collisions**
    - i.e. two threads trying to change the same variable happens only rarely
- The reason for this is hinted at by the word “transactional” in STM
  - Changes to shared mutable state occur during transactions
    - If a transaction fails, updates need to be rolled back
    - You want to avoid the performance hit of rollbacks to maximize concurrency

The Problems

- The concurrency problems being addressed by the STM include
  - **synchronization**
- and
  - the **conflation of identity and state** by imperative OO programming languages

Brief Review: Problems with Synchronization (I)

- With **shared mutability**, there exists the potential for
  - **race conditions**
    - thread A changes the value of X at the same time as thread B
  - **visibility problems**
    - thread A changes the value of X but thread B does not see the change
- To **avoid these problems**, we must **add synchronization**
  - synchronized keyword, synchronized blocks, locks

Brief Review: Problems with Synchronization (II)

- **Adding synchronization** leads to **OTHER** problems
  - programmers can get synchronization **wrong**
    - they can be **too conservative** and **force performance back to single-threaded levels**
    - **race conditions can lurk**
  - once synchronization has been put in place
    - **threads slow down as contention occur**
      - i.e. threads that want to access the same lock at the same time
    - **deadlock** can occur, as well as **live lock** and **starvation**

Conflating Identity with State (I)

- In OOP, when we create a new instance of a class
  - we receive a pointer to the instance that serves as both
    - **its identity** (this instance represents Ken the Employee)
    - **its state** (this instance shows that Ken started work in July 1998)
- This merging of identity and state is a natural consequence of
  - using classes to combine state and behavior
  - having classes encapsulate (or hide) state behind a set of methods

Conflating Identity with State (II)

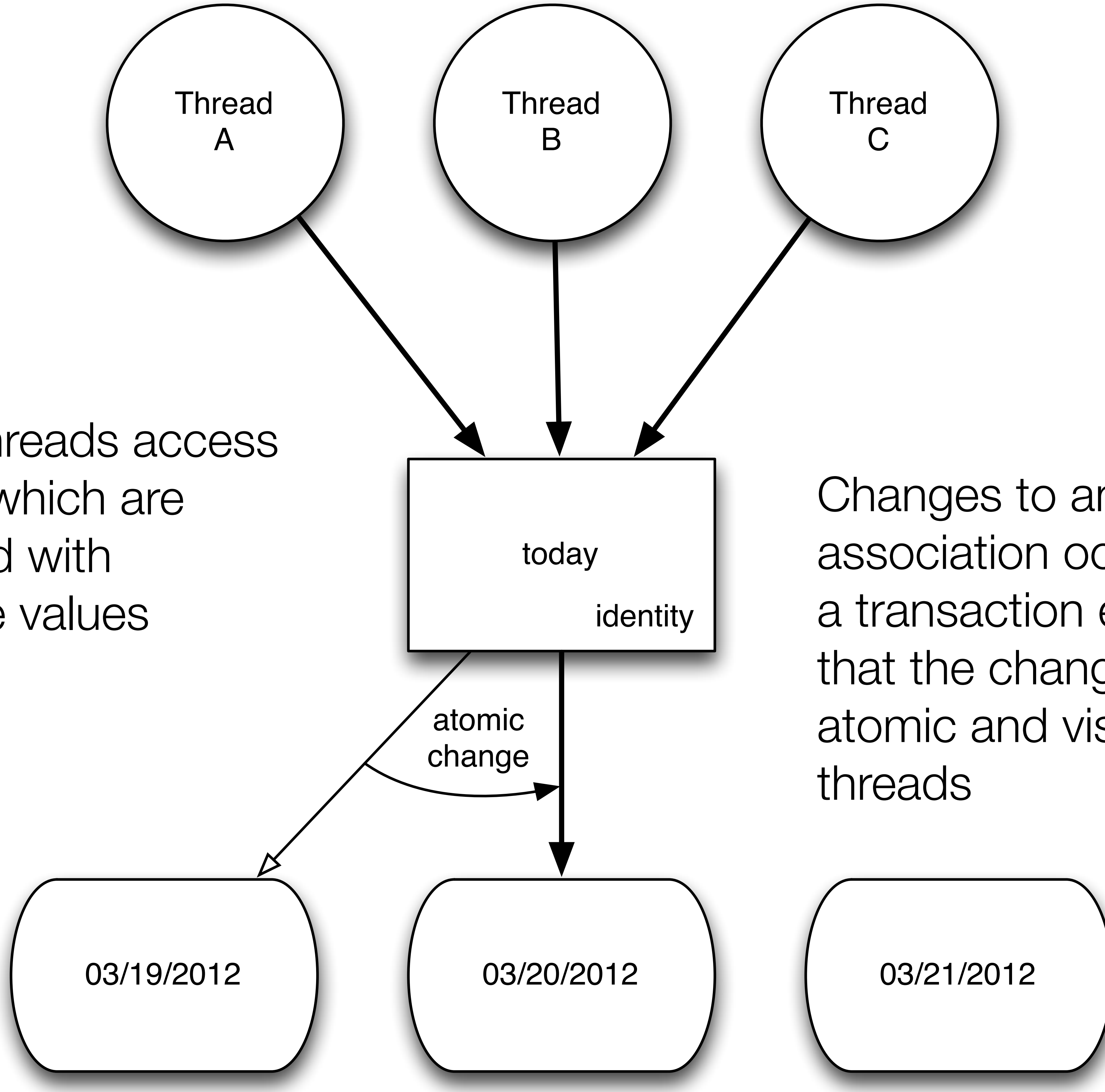
- This **merging of state and identity leads to problems**
  - anyone with access to Ken the Employee can change his start date
  - the previous start date is lost forever
  - indeed, there is no indication that Ken's start date was ever anything else
  - and since Ken actually started in July 1998, the new start date is wrong
- In concurrent situations, a thread with a pointer to **Ken the Employee** has to assume that Ken's state **can change at any moment**
  - and thus the thread is forced to use synchronization to block access to Ken the Employee by other threads while we work with Ken the Employee

The (old) model is wrong

- Rather than having identity and state merged, **the two must be separated**
- In this new model, **identity** is defined as
  - a stable logical entity associated with a series of different values over time
- A **value** is defined as
  - something that doesn't change. All values are **immutable**
- Identity ≠ Name
  - Thread A can point to "Ken the Employee" with a variable called "ken"; Thread B can point to "Ken the Employee" with a variable called "father"
  - Ken and Father are names for the same identity

Values Do Not Change

- The Date “August 25, 1968” **never changes**
- You might have an identity called “today”
  - At one point, “today” was associated with “August 25, 1968”
  - The next day, it was associated with “August 26, 1968” and now that identity is associated with “March 20, 2012”
- The identity is ALWAYS associated with a single immutable value at a given time; someone (a thread) can request that it be associated with a different immutable value (perhaps creating the new value based on the old value)
  - the **association is then changed**, **not the values**
- This immutability is good in concurrent situations, since there is never any danger of a value changing out from under you



In STM, threads access identities which are associated with immutable values

Changes to an identity's association occur within a transaction ensuring that the change is atomic and visible to all threads

Benefits

- Separating identity from state in **concurrent systems** enables
  - **lock-free programming**, and
  - **improved concurrency**
    - because **contention** is **reduced to the bare minimum**
- How? Via transactions
  - **All updates occur via a transaction**
    - if only **transaction A** is updating **identity B**, **no locks are encountered**
    - if **transactions A and B** are updating **identity C at the same time**
      - then the fastest one "wins" and the other is **rolled back and retried**

STM = This New Model

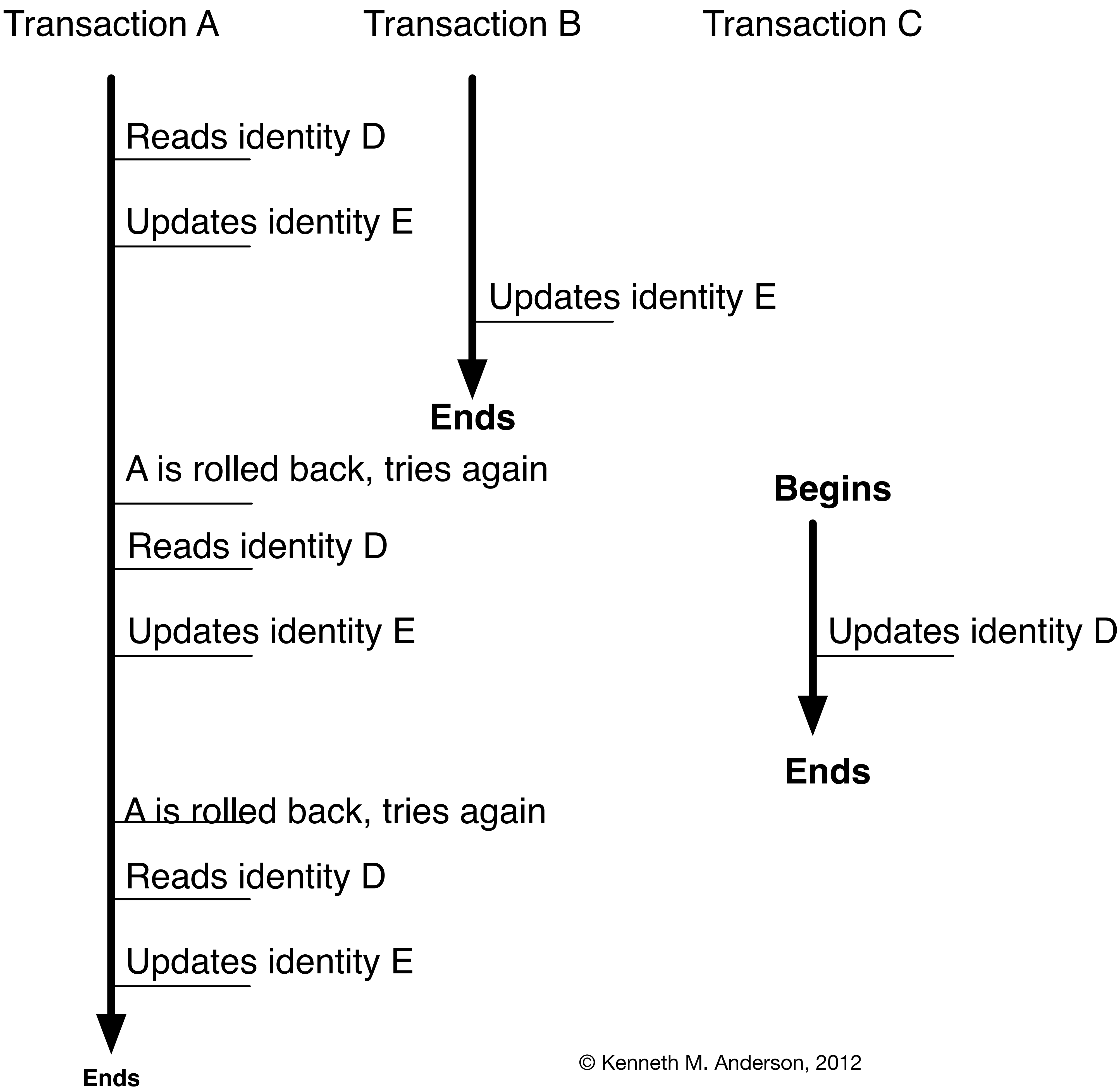
- Software transactional memory enables this new model of
  - **separating identity from state**
- We **tell** the STM **when we have a new identity to track**
  - **providing** the identity with **an initial immutable value**
- Multiple threads can **read this value with no contention**
  - Any request to read the value of an identity simply returns the current value
  - Non-blocking reads help to improve overall concurrency
- When an identity switches to a new value, **which happens atomically**, all subsequent reads get the new value

STM Basics (I)

- STM solves two major problems in the design of concurrent software systems
  - **crossing the memory barrier** (visibility)
  - **preventing race conditions** (consistent state between threads)
- Transactions **ensure** that **changes to identities cross the memory barrier**
  - when those **changes are committed** at the **end of a transaction**
- Within transaction A, the **values of all identities** referenced by transaction A
  - are **guaranteed** to **reflect all changes**
  - of **all transactions** that **completed** before transaction A **begins**

STM Basics (II)

- **Changes within** a transaction **are local** to the transaction
  - that is, **visible only within the transaction**
- **until they are committed**
- If the STM discovers that transaction B has committed a change to identity C while transaction A is also changing C
  - then A is **rolled back**, it receives the latest value of C, and **tries again**
- The STM can also rollback transaction A if A reads from identity D and D is changed by another transaction before A is finished
  - This prevents A from taking actions based on a stale view of the world



Transaction A
Transaction B
Transaction C
Reads identity D
Updates identity E
Updates identity E
Ends
A is rolled back, tries again
Begins
Reads identity D
Updates identity E
Updates identity D
Ends
A is rolled back, tries again
Reads identity D
Updates identity E
Ends

Installing Clojure

- The next few examples make use of Clojure, a recently created programming language that is hosted on the JVM
- To install Clojure
  - Download Cljr at <http://joyofclojure.com/cljr/>
  - Locate the cljr-installer.jar file downloaded as a result of step 1
  - Run “java -jar cljr-installer.jar”
  - Add $HOME/.cljr/bin to your PATH
  - Once that is done, test your installation: “cljr help”
  - If that works, try “cljr repl”; if all goes well, you will be presented with a command prompt that accepts Clojure forms

Updates and Transactions (I)

- If the value of an identity (known as a “ref” in Clojure) is updated outside of a transaction, then an exception is thrown
  - ```
    (def balance (ref 0))
    (println "Balance is" @balance)
    (ref-set balance 100)
    (println "Balance is now" @balance)
    ```
  - This fails with an IllegalStateException; **DEMO**
- Clojure is a Lisp-based language built on the JVM
  - **def** is a function used to create **bindings** between a **symbol** and a **value**
    - The first line creates a **symbol** called **balance** that points at a **mutable identity** with an **immutable value** of 0;
    - @ is the “deref” operator. It follows the association to get the ref’s value
    ```

Updates and Transactions (II)

- To create a transaction, we must wrap code that changes a ref with a call to the function **dosync**
  - ```
    (def balance (ref 0))
    (println "Balance is" @balance)
    (dosync
       (ref-set balance 100))
    (println "Balance is now" @balance)
    ```
  - This time the change is applied and the 2nd println shows the new value
- This is hardly surprising; in this simple program, we do not have other threads running that have access to balance
  - Otherwise, we might find ourselves in a situation where the call to dosync fails and our transaction is rolled back and tried again
    ```

Increment Revisited (I)

- Recall back in Lecture 4, we demoed a program that
  - launched a bunch of threads (10)
  - that incremented a shared variable a number of times (3)
- At the time, we demonstrated that the threads "stomped" on the variable
  - The final value of the variable was much less than 30
- We then showed how we could protect the variable by using the synchronized keyword
- Here's the same program using Clojure and Software Transactional Memory

Increment Revisited (II)

- First, we need a ref to represent the shared integer variable
  - `(def mycount (ref 0))`
- This creates a ref called mycount and sets its initial value to 0
- Second, we need a vector to store references to our worker threads
  - `(def workers (atom []))`
- We create an empty Clojure vector: []
  - And indicate that we'll be updating it: (atom [])
  - A Clojure **atom** is another "reference type" or "identity" that has an association with a value that can change over time
    - We will use the **swap!** function to swap the current value for a new value

Increment Revisited (III)

- Third, we need a function that will be executed by a worker thread
  - This is our “task”
- `(defn worker [id]`
  - `(dotimes [x 300]`
    - `(dosync`
      - `(alter mycount inc))`
    - `(println (str “worker ” id “: increment ” x))))`
- defn creates a function, in this case called worker, which accepts a single argument, its id; all Clojure functions implement java.util.concurrent.Callable!
- It creates a transaction (dosync) and increments the value of mycount

Increment Revisited (IV)

- Fourth, we need a function to launch all of our worker threads
  - `(defn launch []`
    - `(dotimes [x 10]`
      - `(swap! workers conj (future (worker x)))))`
- This creates a function launch with zero arguments
  - It creates 10 worker threads by calling (future (worker x))
  - (future (worker x)) invokes the function “worker” on a separate thread and returns a future (behind the scenes a java.util.concurrent.Future!) that we store in our vector by “conjoing” (conj) the future onto the vector
    - swap! is used to update our “workers” atom with the new vector

Increment Revisited (V)

- Fifth, we are now ready to launch the workers, wait for them to be done, and print out the final value of mycount
    - All of this happens on the main thread
- `(launch)`
- `(doseq [w @workers]`
    - `(deref w))`
- `(println “Final count: ” @mycount)`
- The call to doseq loops over the workers and “deref”s them
    - This is equivalent to calling get() on java.lang.concurrent.Future
    - The main thread blocks on each worker thread until they are all done

Operations That Change Refs (I)

- We've now seen two examples of functions that can change the value of a ref inside of a transaction
  - **ref-set**: sets the value of the ref (identity)
  - **alter**
    - takes a function f and applies it to the current value of the ref
    - the in-transaction value of the ref becomes the value returned by f
    - this might happen several times during a transaction
    - the last value of the ref is committed at the end of the transaction
    - the new value is now visible to other threads

Operations That Change Refs (II)

- The last function that can change the value of a ref inside of a transaction
  - commute
    - takes a function f and applies it to the current value of the identity
    - the in-transaction value of the ref becomes the value returned by f
    - then, just as the transaction is committed, we check to see if some other transaction has changed this ref
    - if so, **rather than having the transaction fail**, we get the most recent value, **apply our function again**, and **commit that value instead**
- Use commute when **you do not care about the order in which your transactions commit**
  - for instance, updating an integer or adding items to an **unsorted** collection (it doesn't matter whether "ken" or "max" is added to a set first)

ACI not ACID

- STM Transactions are like database transactions (minus durability)
  - Atomicity: STM Transactions are atomic
    - all changes get committed (and become visible) or none at all
  - Consistency:
    - if multiple transactions are running and all of them complete, then the change to the system is consistent with the cumulative effect of their actions
  - Isolation
    - transactions do not see partial changes of other transactions, changes only become visible once a transaction successfully completes

How is this implemented? (I)

- Clojure's STM uses Multiversion Concurrency Control similar to what is found in databases
  - The basic strategy is one of optimistic locking
    - We do not pause to take out a lock on the items we want to change because we are **optimistic** that we can **change them without contention**
- At the start of a transaction, all refs that we access are copied
  - We then make changes to the copies
  - If any of our refs do get changed by other transactions, our copies are discarded and we try again (until we succeed or a max_retry_limit is reached)
  - Otherwise are copies are written to memory when the transaction commits

How is this implemented? (II)

- The gory details
  - <http://java.ociweb.com/mark/stm/article.html>

Examples (I)

- The book provides several examples of STM in action
  - concurrentChangeToBalance
    - one balance, two transactions (debit and credit);
    - code is designed to trigger a collision between the transactions
    - as a result, one transaction fails and is retried
  - concurrentListChange
    - two transactions update a list; original list is immutable and a binding to it does not change; the ref however points to the updated list

Examples (II)

- The book provides several examples of STM in action
  - writeSkew and noWriteSkew
    - two updates to a balance cause a property to be violated
    - this occurred because the transactions did not track changes to a ref that is only accessed not updated during the transaction
    - to fix, you need to pass that reference to the function **ensure** which then **monitors changes to that read-only ref** and will **cause the current transaction to fail if that ref changes** during the life of the transaction

Moving beyond Clojure

- Clojure was NOT the first language to provide access to STM-based concurrency
  - It did help to popularize STM by baking it directly into the language
    - Languages that do not support it directly must use frameworks
- There are several options available to use STM in other languages
  - Chapter 7 looks at STM in Groovy, Java, JRuby, and Scala
- For Java, options include
  - using Clojure from within Java (Java can call Clojure and vice versa)
  - Multiverse is a Java-based implementation of STM
  - Akka is a Scala-based framework that internally makes use of Multiverse

Installing Akka

- Akka can be retrieved at
  - <http://akka.io/downloads/>
- In particular, download
  - <http://download.akka.io/downloads/akka-microkernel-1.3.1.zip>
- Unpack the zip file and put it in a dir;
  - that location becomes AKKA_HOME
- On the next slide are instructions for MacOS X and Linux users; Windows users will need to look for instructions on-line

Installing Akka (II)

- For MacOS X/Linux under bash, you can edit your .bash_profile to include something like this
  - export AKKA_HOME=/Path/to/akka-microkernel-1.3.1/dir/
  - export AKKA_JARS="$AKKA_HOME/lib/scala-library.jar"
  - export AKKA_JARS="$AKKA_JARS:$AKKA_HOME/lib/akka-stm-1.3.1.jar"
  - export AKKA_JARS="$AKKA_JARS:$AKKA_HOME/lib/akka-actor-1.3.1.jar"
  - export AKKA_JARS="$AKKA_JARS:$AKKA_HOME/lib/multiverse-alpha-0.6.2.jar"
  - export AKKA_JARS="$AKKA_JARS:$AKKA_HOME/config"
  - export AKKA_JARS= "$AKKA_JARS:."
- We will only need these jars to compile/run the examples from the book

More on Akka (I)

- Akka provides us with Java APIs that enable STM
- In particular, Akka refs act similar to Clojure refs with one main distinction
    - We can update a Akka ref outside of a transaction
        - Such updates are wrapped in a transaction automatically
    - Akka refs are created using the type akka.stm.Ref<T>
- Otherwise, we programmatically create transactions and then update refs within them; their behavior is then identical to what we saw with Clojure
    - Akka adds the notion of nested transactions. As a result, we can be in a transaction and call methods that in turn create transactions
    - Akka will ensure that all such transactions complete before the outer transaction can complete

More on Akka (II)

- When we have a reference to an Akka ref, we can
  - retrieve its value with a call to get()
  - update its value with a call to swap()
  - Both of these calls will create a transaction behind the scenes if we do not call them from within the context of a transaction
- To run code in a transaction, we create an anonymous instance of the Atomic<T> class and insert the code to run in a transaction within a call to the method <T> atomically().
  - We'll see an example in a minute

Example: Return to Energy Source

- The book updates the EnergySource example from chapter 5 to make use of Akka's implementation of STM
    - **DEMO**
- To compile the demo, you will use this command
    - javac -classpath $AKKA_JARS *.java
- To run the demo, you will use this command
    - java -classpath $AKKA_JARS useEnergySource
    - java -classpath $AKKA_JARS Main
- The latter runs a slightly modified version of my own EnergySource using program that we discussed during Lecture 12

Discussion (I)

- Changes
  - All of the internal instance methods converted to be Akka Refs
  - Since we can now trust that the value of the keepRunning flag will be
    - both consistent and visible (due to Akka transactions)
  - we change the way the replenish task is handled;
    - synchronized goes away on methods;
    - keepRunning.get() and keepRunning.swap() used instead
  - All other updates (including updating both level and usage) are handled atomically via transactions; no locking required!

Summary

- In this lecture, we introduced the approach to concurrency known as the software transactional memory
  - Transactions are used to update shared mutable state (refs) with guaranteed consistency and visibility
  - Had to change our notion of “state” to make this possible
    - State and Identity are no longer conflated
    - Instead, identities maintain associations with immutable state over time
  - Transactions are optimistic that contention with other threads will not be an issue
    - they make changes with no locking and then fail if contention occurred

Coming Up Next

- Lecture 20: More examples of STM in Java and other languages
- Lecture 21: Agile Project Execution