Clojure Concurrency Constructs

CSCI 5828: Foundations of Software Engineering
Lecture 12 — 10/02/2014

Goals

- Cover the material presented in Chapters 3 & 4 of our concurrency textbook
  - Books examples from Chapter 3: Day 2 and Day 3
    - reducers, futurers, promises
  - Begin coverage of material presented in Chapter 4

- Supplemental Resources
  - <http://clojure.com/blog/2012/05/08/reducers-a-library-and-model-for-collection-processing.html>

Reducers

- Reducers is a library provided with Clojure to provide
  - “an alternative approach to using sequences to manipulate standard Clojure collections. Sequence functions are typically applied lazily, in order, create intermediate results, and in a single thread.
  - A reducer is the combination of a reducible collection (a collection that knows how to reduce itself) with a reducing function (the "recipe" for what needs to be done during the reduction).
  - The standard sequence operations are replaced with new versions that do not perform the operation but merely transform the reducing function. Execution of the operations is deferred until the final reduction is performed.
  - This removes the intermediate results and lazy evaluation seen with sequences.” — Taken from <http://clojure.org/reducers>

Reducers explained

- Essentially, reducers allow you to specify a sequence of operations to be applied to a collection
  - As the operations are applied, no work is performed
    - instead each operation manipulates the reducing function, transforming what it will do when it gets invoked
    - When reduce or fold is finally applied to the collection,
    - the operation of the reducing function is performed with as much parallelism as can be accomplished given the size of the data set and the transformations to be performed
- Since lazy sequences are not used, the following notes appear in the Reducers library documentation
  - Use reducers when the source data **can be generated and held in memory**, the work to be performed is **computation bound**, the amount of data is **large**

Operations

- The following Reducers operations transform the reducer function
  - r/map, r/mapcat, r/filter, r/remove, r/flatten, r/take-while, r/take, r/drop
- To invoke processing, of a sequence of these operations
  - r/reduce or r/fold
- To create a collection of the final result
  - use r/foldcat or the into function from the standard Clojure library
- When you call r/fold or r/foldcat, it does the following
  - It partitions the underlying data set into n groups of 512 elements
  - Applies reduce to each of the groups
  - Recursively combine each partition using Java's fork/join framework

Simple Examples (from the book)

- To gain access to the reducers library
  - `(require '[clojure.core.reducers :as r])`
- `(r/map (partial * 2) [1 2 3 4])`
  - If you call this, you do not get back a collection, instead you have created an initial “reducible” that can be combined with other reducibles until r/reduce or r/fold is called
- `(into [] (r/map (partial + 1) (r/filter even? (range 1000000)))`
  - This produces a 500,000 collection of odd numbers from 1 to 999999
  - The important thing to realize is that `into` calls `reduce` in the background and when it does the reducing function is set-up to only allow even numbers through and to add one to all such numbers
    - Both operations are applied at once to the members of each partition

Performance

- Out of curiosity, I executed this form with 4GB of memory allocated for the JVM
  - `(r/foldcat (r/map (partial + 1) (r/filter even? (range 100000000)))`
- This creates a collection of 100M integers, throws out all odd integers, and adds one to the resulting numbers
  - I was curious too see how much concurrency r/foldcat achieved while performing this operation
- The answer?
  - Not much; only 100% CPU utilization
- Why?
  - We didn't meet all three preconditions => the combination of even? and increment doesn't take very long and so it isn't compute bound

The Book's Detour

- The book spends a LONG time in the book trying to teach you HOW the reducers framework works rather than showing how to use it
    - I'm going to skip the material that tries to recreate the reducers framework and instead show you an example where it can help speed up single threaded code
- We'll do that with a program that counts the number of prime numbers that appear within a given range
    - We'll start with a single threaded version of the code and time it
        - We'll then make it parallel with the reducers framework and see how much of a speed up we get

Counting Primes

- Our program will have the following constraints
  - We will never create a lazy sequence
    - Reducers need their collections in memory
    - As a result, we'll give a generous amount of memory to the JVM
- Our program will be designed in the following way
  - We will create a collection that contains numbers from 1 to n
  - We will apply map to this function using a function prime?
  - We will apply map again using a function to convert true to 1 and false to 0
  - We will apply reduce to this final collection to count all the ones
    - This sum will be our final answer

Our prime? function

- Steal from the best
  - I borrowed this prime? function from Daniel Gruszczyk
    - He did the hard work of developing a version of prime? that is as fast as corresponding, heavily optimized Java code
    - See <http://blog.programmingdan.com/?p=35> for details

prime? <http://blog.programmingdan.com/?p=35>

```
(defn prime? [n]
  (let [sqrt (Math/sqrt n)]
    (cond
      (< n 2) false
      (= n 2) true
      (even? n) false
      :else
        (loop [i 3]
          (cond
            (> i sqrt) true
            (zero? (unchecked-remainder-int n i)) false
            (< i sqrt) (recur (+ i 2)))))))
```

- For any n,
  - if n < 2, return false; if n == 2, return true; if n is an even number, return false; otherwise loop from 3 to the square_root of n by 2. If the remainder of n by the current integer is zero, return false; otherwise return true

Single-Threaded `count-primes` Function

- We will now build up the pieces we need for the single-threaded version of our count-primes program
  - The trick is to make sure that no lazy sequences are created
  - Our program will consist of functions that perform each of the four steps outlined on slide 9
    - produce-range, find-primes, convert-to-numbers, sum

produce-range

- `(defn produce-range [n]`
  - `(into [] (range n)))`
- The use of into forces range to create a collection with all of the elements in it
  - Otherwise, range produces a lazy sequence

find-primes

- `(defn find-primes [numbers]`
  - `(doall (map prime? numbers)))`
- The call to doall forces the lazy sequence produced by map to be fully realized
  - We do not pass back a lazy sequence
  - We pass back a full in-memory sequence of true/false values

convert-to-numbers

```
(defn convert-to-numbers [primes]
  (doall (map (fn [x] (if (true? x) 1 0)) primes)))
```

- Again the call to doall forces the lazy sequence to be fully realized
- In this case, we pass an anonymous function to map
- If that function finds a “true”, it replaces it with 1 otherwise it replaces it with 0

count-primes

- Putting this all together we get

```
(defn count-primes [n]
  (sum (convert-to-numbers (find-primes (produce-range n)))))
```

- It is remarkably clean code
  - produce the range,
  - find the primes,
  - convert to 1s and 0s
  - and compute the sum!

Performance

- Number of primes from 1 to 10M: 664,579
  - `(time (count-primes 10000000))`
    - averages 6.4 seconds; 100% CPU utilization
- Number of primes from 1 to 100M: 5,761,455
  - `(time (count-primes 100000000))`
    - required 6.67 GB of memory!
    - averages 140 seconds (2.3 minutes); 100% CPU utilization

Parallel version of count-primes using Reducers

- Our parallel version of count-primes involves the following
  - `produce-range` stays the same: we want an in-memory collection
  - `find-primes` gets converted to
    - `(defn find-primes-par [numbers]`
      - `(r/map prime? numbers))`
  - `convert-to-numbers` gets converted to
    - `(defn convert-to-numbers-par [primes]`
      - `(r/map (fn [x] (if (true? x) 1 0)) primes))`
  - `sum` gets converted to
    - `(defn sum-par [numbers]`
      - `(r/fold + numbers))`
- The first two produce reducibles, the third performs the entire thing in parallel

The parallel version of count-primes

- The parallel version of count-primes simply calls our new functions
  - `(defn count-primes-par [n]`
    - `(sum-par (convert-to-numbers-par (find-primes-par (produce-range n)))))`
- Performance?
  - Number of primes from 1 to 10M: 664,579
    - `(time (count-primes-par 10000000))`
      - averages 2.5 seconds; ?? % CPU utilization (to fast to see!)
      - almost 3 times faster!
  - Number of primes from 1 to 100M: 5,761,455
    - `(time (count-primes 100000000))`
      - averages 47 seconds; as high as 790% CPU utilization; 3x speedup!
      - only 3.5 GB of memory (only one big collection made, not three)

Discussion

- Counting primes is more compute bound than our previous examples
  - and as such, we indeed saw a speed-up using reducers
  - and the approach to concurrency was typical of the functional programming approach
    - code up a sequential version
    - swap functions as needed to enable a parallel version
- Our only constraint with the reducers library is that our collections need to be able to fit in memory

Futures

- Our book next looks at two related concepts: Futures and Promises
  - They are concurrency constructs for functional programs
- **Futures** are created by passing a block of code to the function `future`
  - That block of code will at some point execute in a separate thread
- The code that calls `future` is not blocked
  - The call to `future` returns immediately returning a handle to a future object
- To retrieve the value of the future, you dereference the future object
  - If the future is still running in the other thread, the calling thread will block; otherwise it gets the last computed value of the future

Example: Using futures

- `(def sum (future (+ 1 2 3 4 5 6 7 8 9 10)))`
- `sum` is a symbol that points at a future object
  - The code `(+ 1 2 3 4 5 6 7 8 9 10)` runs in a separate thread
- If we ever want to know the value of `sum`, we have to deference it
  - `@sum` or `(deref sum)`: both return 55

Promises

- A promise is similar to a future with one key difference
  - It produces an object that will at some point provide a value
  - If you `deref` the promise and the value is not available, you block
  - No code gets executed when creating a promise; instead some other thread has to compute the value of the promise and then `deliver` that value
- If some other thread was blocked when `deliver` is called, then that thread unblocks and receives the generated value

Example: Using Promises

- Create a promise
  - `(def meaning-of-life (promise))`
- Create a future that blocks on this promise
  - `(future (println "The meaning of life is:" @meaning-of-life))`
- This creates a thread that immediately blocks trying to get access to the value associated with the promise
- Deliver the value
  - `(deliver meaning-of-life 42)`
- In the REPL, immediately after this call to deliver, the thread created by `future` unblocks and prints `"The meaning of life is: 42"`

Simple Web App (I)

- The book provides an interesting example of using future, promise, and deliver to implement a simple web service
    - The web service's goal is to print out snippets of text which have some pre-defined order (in the example, the lines of a poem)
    - The snippets can arrive in any order
        - but the web service guarantees that it will print the snippets out in order
- The web service exposes one endpoint: /snippet/:n
    - The number of the snippet is embedded in the URL. The text of the snippet is provided in the body of an HTTP PUT request

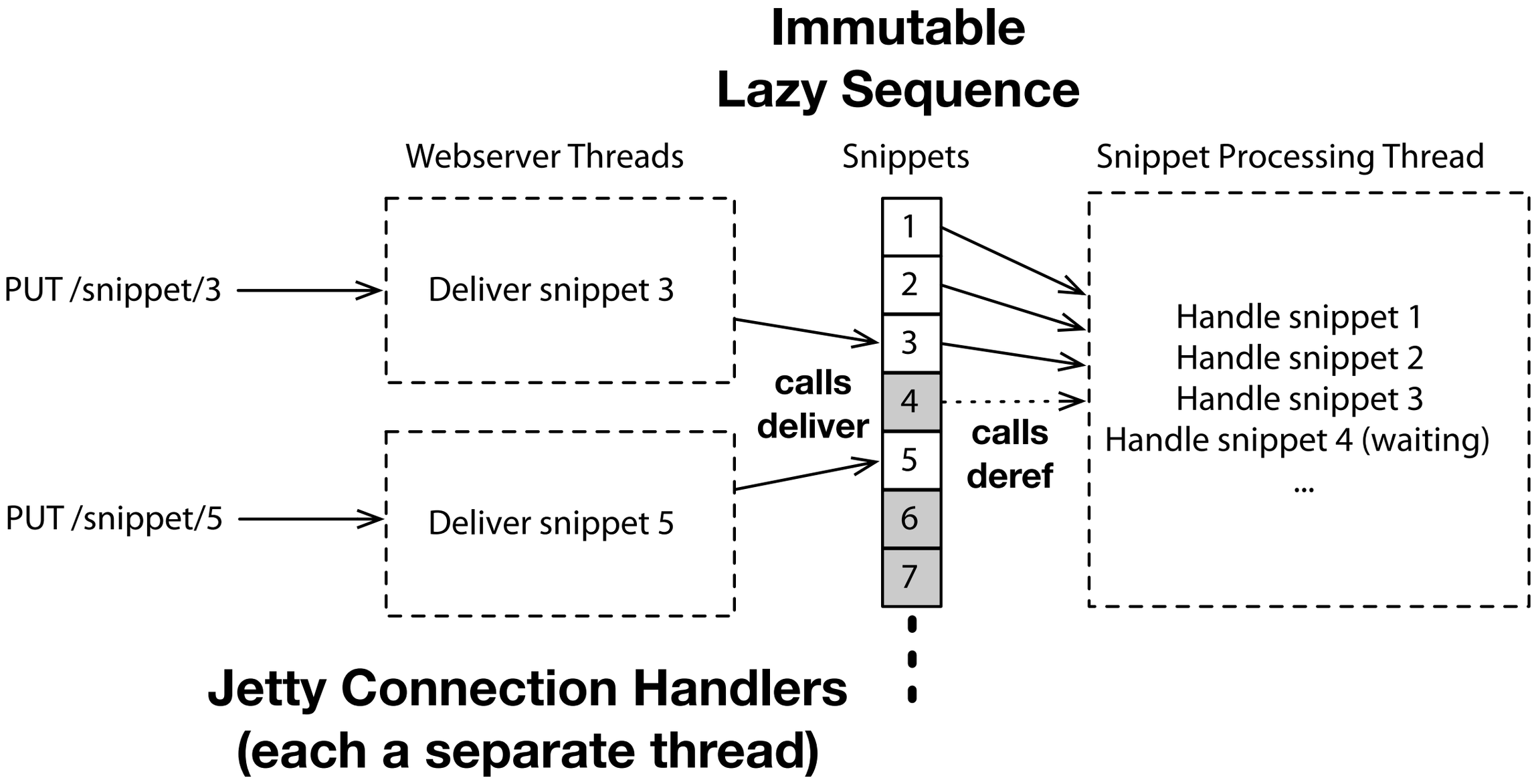
Simple Web App (II)

- The set-up
  - `(def snippets (repeatedly promise))`
- `repeatedly` creates an infinite lazy sequence. Any time a new element is needed, it calls the function that was passed to it. `promise` returns a promise object as usual
- `(future`
  - `(doseq [snippet (map deref snippets)]`
  - `(println snippet)))`
- The call to `future` creates a new thread. In that thread, `doseq` is used to ask for all elements of the lazy sequence. This means that `map` will ask for an element, apply `deref` to it, and put the result in an output collection
  - Each result is also bound to `snippet` and printed; If a snippet hasn't arrived, then the call to `deref` causes this thread to block

Simple Web App (III)

- Meanwhile, the web server (jetty) is waiting for connections from clients
  - when it receives a connection at the /snippet URL, it asks one of its connection handlers (i.e. a thread) to handle the request
  - It takes the snippet number from the URL and the text from the body of the request and calls this function
    - `(defn accept-snippet [n text] (deliver (nth snippets n) text))`
- `(nth snippets n)` retrieves the nth promise object from the sequence
  - Those elements will be created if needed, otherwise the previously created promise object will be returned
  - It then calls `deliver` on that promise object giving it the text as its value
    - If the thread created by `future` is blocked on that promise object, it finally unblocks, prints the text, and calls `deref` on the next promise

Simple Web App (IV)



Less Simple Web App

- The book goes on to enhance the web app in various ways
  - adding support for sessions
  - adding the ability to translate snippets to a different language
    - Great use of a **future** demonstrated with this enhancement
      - The future is used to wrap a call to another web service
      - The calling code creates the future and then derefs it
        - This makes it block until a value has been returned, which means it automatically waits until the web service call has been made and returns a response
- However, I'm not going to spend more time in lecture on this example

Clojure's Mutable Variables

- Clojure is an **impure** functional language
    - Pure functional languages provide no support for mutable data
- Clojure however provides a number of different types of **concurrency-aware** mutable variables, each tailored to a different type of update
    - refs — synchronous, coordinated updates (software transactional memory)
    - atoms —uncoordinated, synchronous updates
    - agents —asynchronous updates
- All of these are supported by Clojure's use of persistent data structures
    - We will look at atoms and persistent data structures in the remainder of this lecture; we'll look at refs, STM, and agents in future lectures

Atoms

- Atoms in Clojure are atomic variables, similar in concept to the classes provided by java.util.concurent.Atomic (such as AtomicInteger)
  - Updates to the value of these variables are synchronous
    - locks are not used so contention is minimal or nonexistent
      - threads may discover that the value has been updated by another thread in between it's attempt to read the value and its attempt to write the value; if that's the case, its value is updated and it tries again
    - all updates are guaranteed to cross the memory barrier
- Updates to atoms are "uncoordinated", what this means is that if you have two or more atoms that you want to change, you can't change them both in an atomic way; if you want to do that, you need to use refs

Using Atoms

- To create an atom
  - `(atom <initial-value>)`
  - Example: `(def counter (atom 0))`
- To get an atom's value
  - `(deref counter)` or `@counter`
- To update an atom's value
  - `(swap! <atom> <function> <args>)`
  - `(swap! counter inc)`
- To reset an atom's value
  - `(reset! <atom> <newvalue>)`
  - `(reset! counter 0)`

Counter example with Atoms (I)

- One atom, counter, holds an integer value
  - `(def counter (atom 0))`
- One watcher on counter
  - `(add-watch counter :print #(println "Changed from " %3 " to " %4))`
- One atom, log, holds a vector
  - `(def log (atom []))`
- One promise
  - `(def wait-for-it (promise))`
- Two futures
  - `(def t1 (future (perform-updates 1)))`
  - `(def t2 (future (perform-updates 2)))`

Counter example with Atoms (II)

- Each future executes this function
  - `(defn perform-updates [i]`
    - `(deref wait-for-it)`
    - `(doseq [x (range 20)] (perform-update i)))`
- By calling deref as the first thing they do, we block both threads
- To unblock them, all we need to do is deliver a value to the promise
  - `(defn wake-them-up []`
  - `  (deliver wait-for-it "Go for it!"))`
- To check on results, we can just deref the atoms
  - `@counter` and `@log`

Counter example with Atoms (III)

- We update the atoms with the perform-update function
  - `(defn perform-update [i]`
    - `(let [msg (log-message i)]`
      - `(swap! log conj msg)`
      - `(swap! counter inc)))`
- The calls to swap! will update the atoms atomically but NOT in a coordinated fashion (as we will see)
- The last piece of the puzzle, log-message generates the following string
- `(defn log-message [i]`
  - `(format "Thread %d updating atom: %d to %d" i @counter (inc @counter)))`

The Results

- @counter => 40

- @log => ["Thread 1 updating atom: 0 to 1" "Thread 2 updating atom: 0 to 1" "Thread 2 updating atom: 2 to 3" "Thread 1 updating atom: 2 to 3" "Thread 2 updating atom: 3 to 4" "Thread 1 updating atom: 4 to 5" "Thread 2 updating atom: 5 to 6" "Thread 1 updating atom: 6 to 7" "Thread 2 updating atom: 7 to 8" "Thread 1 updating atom: 8 to 10" "Thread 2 updating atom: 9 to 10" "Thread 1 updating atom: 10 to 12" "Thread 2 updating atom: 11 to 12" "Thread 1 updating atom: 13 to 14" "Thread 2 updating atom: 13 to 14" "Thread 1 updating atom: 14 to 15" "Thread 2 updating atom: 15 to 16" "Thread 1 updating atom: 16 to 17" "Thread 2 updating atom: 17 to 18" "Thread 1 updating atom: 18 to 19" "Thread 2 updating atom: 19 to 20" "Thread 1 updating atom: 20 to 21" "Thread 2 updating atom: 21 to 22" "Thread 1 updating atom: 22 to 23" "Thread 2 updating atom: 23 to 24" "Thread 1 updating atom: 24 to 25" "Thread 2 updating atom: 25 to 26" "Thread 1 updating atom: 26 to 27" "Thread 2 updating atom: 27 to 28" "Thread 1 updating atom: 28 to 29" "Thread 2 updating atom: 29 to 30" "Thread 1 updating atom: 30 to 31" "Thread 2 updating atom: 31 to 32" "Thread 1 updating atom: 32 to 33" "Thread 2 updating atom: 33 to 34" "Thread 1 updating atom: 34 to 35" "Thread 2 updating atom: 35 to 36" "Thread 1 updating atom: 36 to 37" "Thread 2 updating atom: 37 to 38" "Thread 1 updating atom: 38 to 39"]

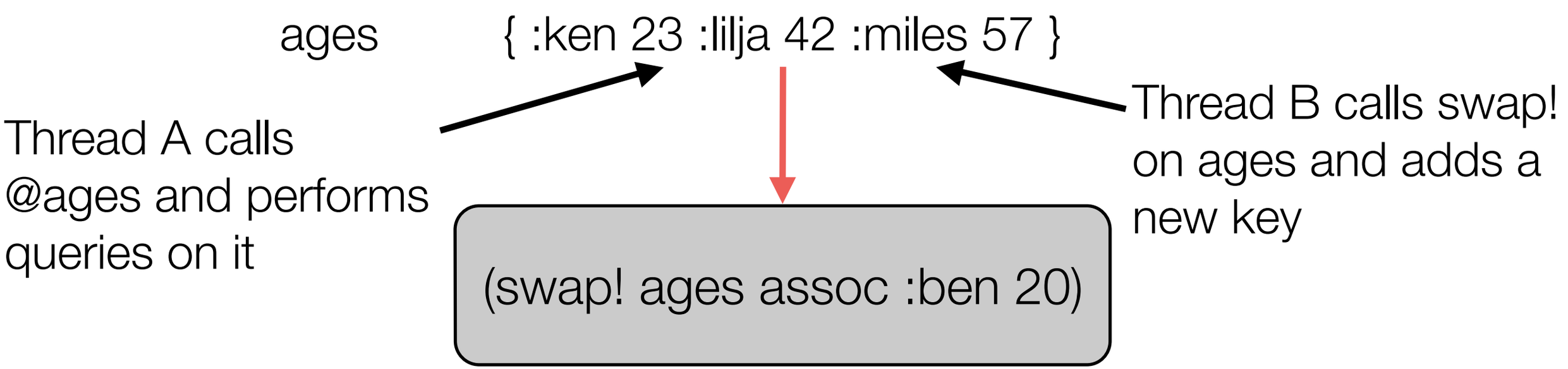
Discussion

- counter ends up with the correct value: 40
- log also ended up with 40 entries but the messages themselves display the interesting behaviors that can occur with reads on an atomic variable that is being updated
  - “Thread 1 updating atom: 8 to 10” (!)
    - Recall, we used the following code to generate this string
    - `(format "Thread %d updating atom: %d to %d" i @counter (inc @counter))`
- So, @counter was 8 when we first called it; it was updated by thread 2 in the background, such that its value was 9 when we deref’d it again
  - This is what we mean by uncoordinated updates, even though we were in the process of updating both atoms in Thread 1, Thread 2 was active in the background, changing the value of @counter
    - No update was lost but there was no coordination

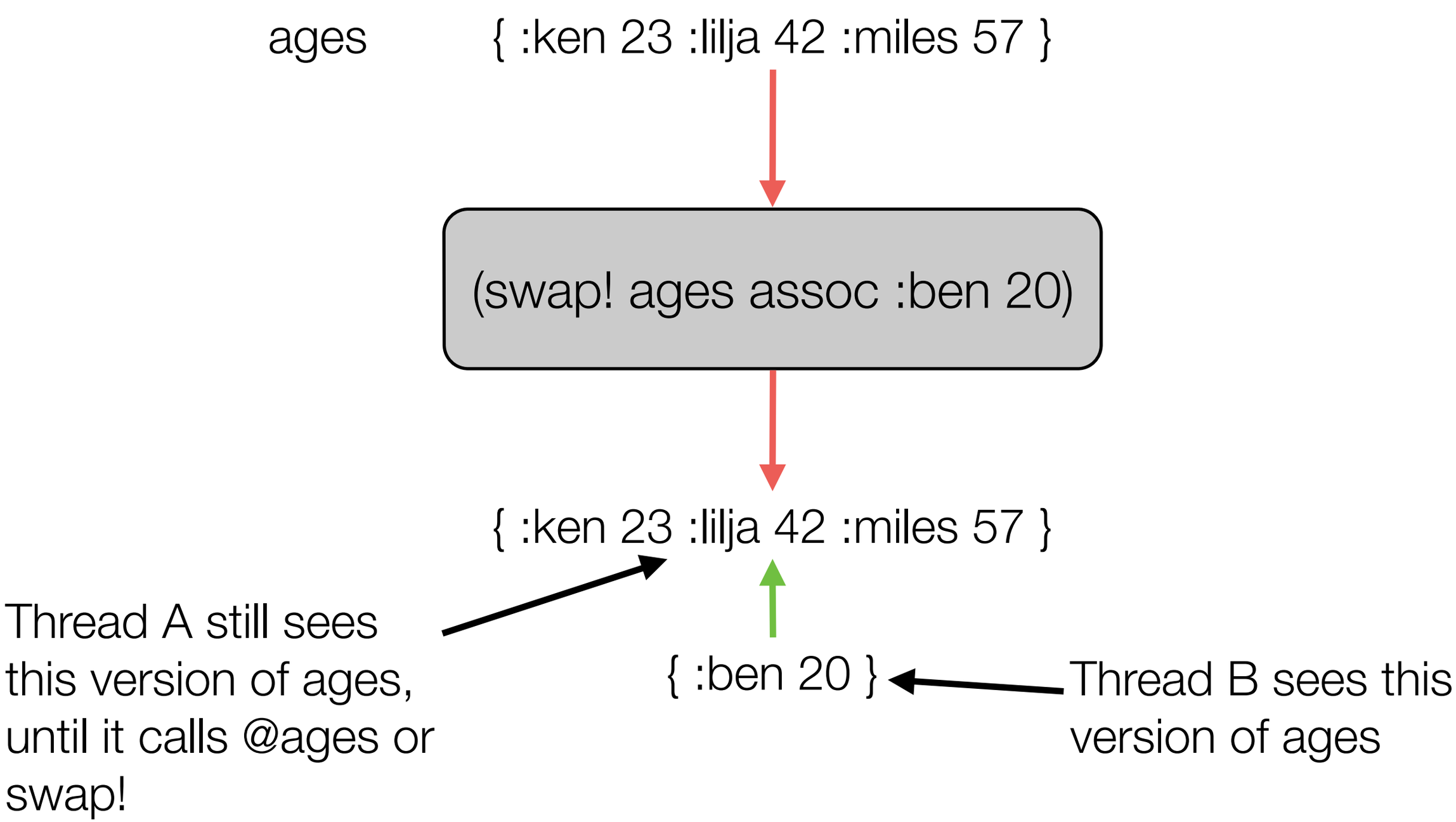
Persistent Data Structures

- All of Clojure's collections are implemented as persistent data structures
    - What this means is that, behind the scenes, when Clojure needs to "make a copy" of a data structure in order to change it, it instead tries to share as much of the previous version of the data structure as it can
- This is important in order to ensure that code accessing data structures from multiple threads maintain a "persistent" view of that structure
    - Say thread A is looking at a map stored in an atom and thread B changes that map
        - Thread A's map will NOT change out from under it, it will have the value it had when it was last read and remain persistent until thread A tries to change the map. At that point, it is informed about the new value and it needs to retry its change within the context of that new value

Example



Example



Discussion

- Note: if A is satisfied with its view of @ages, it never has to update it
  - It will always have access to a data structure that can answer questions about Ken, Lilja, and Miles
- This ability to have threads pointing at different values of a data structure at the same time is referred to as “separating identity from state”
  - Persistent data structures can keep track of all their values over time and allow different threads access to different aspects of that history
    - Each thread points at an immutable data structure that is operational
      - There can be no “stomping” or changing that value underneath it
  - If we need to update to the most recent version, we can
- Think of “ages” as the identity but it’s state can change over time and there’s no reason why we have to update everyone’s view of that state at the same time

Summary

- In this lecture, we've seen a range of Clojure concurrency constructs
  - The reducers library for in-memory, compute bound operations in parallel
  - futures and promises, for inter-thread communication
  - atoms, for uncoordinated, synchronous updates to a single value
- We also talked about persistent data structures which helps to
  - separate the identify of something we want to track in the world from its series of values over time
  - and keep memory concerns low by (behind the scenes) having different versions of the same data structure share as much of the structure of its collection class as we can as changes are applied to it
- This sets us up for refs, software transactional memory, and agents

Coming Up Next

- Lectures 13 and 14: More Clojure concurrency constructs, iteration and release planning, and reviewing for the midterm!
- Lecture 15: **MIDTERM**
- Lecture 16: Midterm Review (if I can swing it)