

Concurrency and Functional Programming

CSCI 5828: Foundations of Software Engineering
Lecture 11 — 09/30/2014

Goals

- Cover the material presented in Chapter 3 of our concurrency textbook
 - Intro to Clojure
 - Books examples from Day 1 and the start of Day 2

Installation

- In order to follow along, you need to install Clojure
 - The best way to do that is with Leiningen
- On Mac OS X with HomeBrew installed, this is easy
 - `brew install leiningen`
- Otherwise, follow the simple instructions on the Leiningen home page
 - <http://leiningen.org>
- The first time you invoke the “lein” script, it will auto-install everything it needs
 - System of Systems: It makes use of Maven in the background to download the packages that it needs!

Use (I)

- For basic use, all you need is the REPL environment
 - `lein repl`
- This loads up a version of Clojure and sets the default name space to “user”
- To write larger systems of functions, you need to adopt the following generic folder structure
 - `project_folder/`
 - `project.clj`
 - `src/`
 - `<project_specific_package_name>/`
 - `<project_specific_file_name>.clj`

Example project.clj

```
(defproject test-prime "1.0"
  :dependencies [[org.clojure/clojure "1.6.0"]]
  :jvm-opts    ["-Xmx4096m"]
  :main test.prime)
```

- This particular project.clj file declares
 - our project is called “test-prime”
 - it has the ambitious version number of “1.0”
 - it wants the Java virtual machine to have up to 4GB of memory and,
 - the main file to load at runtime is “prime.clj” located in src/test/

Use (II)

- With that configuration, you simply enter the directory that contains the project.clj file and type “lein repl”
 - You start in the “test.prime” name space and all of your functions are directly available at the REPL prompt
 - If you ever change your source code and you want to reload all of your functions into an existing REPL session, use the **require** command
 - `(require :reload 'test.prime)`
- That’s all we need to understand with respect to setting Clojure up for initial use; note: there is a LOT more to learn. For instance, if you want to see the source code of a function, you can ask the REPL with this command
 - `(source <function_name>)`
 - **Example:** `(source time)` **or** `(source map)` **or** `(source pmap)`

Clojure

- Clojure is a dialect of Lisp created in 2007 by Rich Hickey
 - It is built on top of the Java Virtual Machine
 - While it is a Lisp, it can make calls into the Java standard libraries
 - Sometimes the answer to “how do you do X in Clojure” is answered with “Just call `java.util...`”, i.e., just use the implementation provided by Java
- Clojure’s design adopts a focus on programming with immutable values and the creation of concurrent programs that are easier to reason about
 - In particular, you can easily find videos of Rich Hickey casting aspersions on concurrent programs with shared mutable state

Clojure and our Textbook

- In Chapter 3, our textbook focuses more on functional programming style and the way that concurrency can be incorporated into functional programming
- It holds off to talk about Clojure's more explicit concurrency constructs
 - atoms
 - persistent data structures
 - agents
 - software transactional memory
- until Chapter 4

Clojure Reference Materials

- The Pragmatic Programmers offer a range of books on Clojure
 - Programming Clojure: <<https://pragprog.com/book/shcloj2/programming-clojure>>
 - Functional Programming Patterns in Scala and Clojure: Write Lean Programs for the JVM: <<https://pragprog.com/book/mbfpp/functional-programming-patterns-in-scala-and-clojure>>
 - Mastering Clojure Macros: Write Cleaner, Faster, Smarter Code: <<https://pragprog.com/book/cjclojure/mastering-clojure-macros>>
- In addition, you can check out the official Clojure website
 - <<http://clojure.org>>
- I also found this (it's out-of-date in places but still good!)
 - <<http://moxleystratton.com/clojure/clojure-tutorial-for-the-non-lisp-programmer>>

Clojure Basics (I)

- Clojure has a fairly basic set of data types (a.k.a forms)
 - Booleans — `true`, `false`
 - Characters — `\a`, `\A`
 - Strings — `"ken anderson"`
 - No value — `nil`
 - Numbers — `1`, `2`, `3.14159`, `0.000001M`, `100000000000000N`
 - Keywords — `:first`, `:last`
 - Symbols — `x`, `i`, `java.lang.String`, `user/foo`
 - Lists — `(1 2 3 4 5)`
 - Vectors — `[1 2 3 4 5]`
 - Sets — `#{1 2 3}`
 - Maps — `{:first "Ken" :last "Anderson"}`
- Note: Commas (,) are whitespace in Clojure. Use them if you want, they will be ignored!

Functions

- If the first element of a list is a symbol that references a function, then the list becomes a function call and will be replaced with its value
 - `(+ 1 2) => 3`
 - `(sort [9 3 5]) => (3 5 9)`
- Functions can be defined using another function called `defn`
 - `(defn name [args*] forms+)`
- The value of the last `form` in `forms+` is the return value of the function
- Anonymous functions can be created as well either with `fn` or shorthand syntax
 - `(fn [x] (+ x 10))`
 - `#(+ x %)`

Symbols

- You can create your own symbols with the function `def`
 - `(def pi 3.14159)`
 - `(def x 10)`
- These statements would add the symbols `pi` and `x` to the current namespace
- The values of these symbols are immutable
 - `(+ x 10) => 20`
- This just references the value of `x`, it doesn't change `x`
- You can run the `def` command again
 - `(def x 5)`
- `x` now has the value 5, but all this command did was **rebind** the symbol

Control Flow

- Control flow structures are just functions
 - `(if (< x 0) "negative" "non-negative")`
 - `(cond`
 - `(< x 10) "small"`
 - `(= x 10) "medium"`
 - `(> x 10) "large"`
 - `:else "uh oh")`
- Loops are a special case
 - there is an explicit `loop` function, but you'll typically avoid it and use `map` and `reduce` instead

Loop (I)

- The generic form of a loop is
 - `(loop [bindings *] exprs*)`
- The call to `loop` creates a “jump point” that allows control to return to the top of the loop by calling the function `recur`
 - `(recur exprs*)`
- The expressions associated with `recur` are allowed to establish new bindings of the symbols created by `loop`
- Let's see an example

Loop (II)

- `(loop [result [] x 5]`
 - `(if (zero? x)`
 - `result`
 - `(recur (conj result x) (dec x))))`
- This expression returns `[5, 4, 3, 2, 1]`
- The bindings at the top initialize `result` to an empty vector and `x` to 5
- The code then checks to see if `x` is equal to 0
 - Since it isn't, `recur` rebinds `result` to be a vector that has the value of `x` appended to it and rebinds `x` to 4
 - The code then jumps back to `loop` and executes again (the initial bindings are then ignored)

Loop (III)

- You can also recur to the start of any function and similarly rebind its parameters
- `(defn countdown [result x]`
 - `(if (zero? x)`
 - `result`
 - `(recur (conj result x) (dec x))))`
- This function will take an input vector and a (hopefully positive) number and appends that number and all of the numbers between it and zero to the vector
 - `(countdown [] 5) => [5 4 3 2 1]`
- The use of `recur` also allows Clojure to use tail recursion, allowing this function to be implemented as a loop and not via recursion

Loop (IV)

- But, this style is rarely needed in functional programming
- Instead, you will use more declarative constructs where the iteration is hidden
 - `(into [] (take 5 (iterate dec 5)))`
 - `(into [] (drop-last (reverse (range 6))))`
 - `(vec (reverse (rest (range 6))))`
- All of these produce the same `[5, 4, 3, 2, 1]` result
- Similarly, you'll use `map` to operate on all members of a list and `reduce` to use all of the members in a list to calculate some value
 - `(map inc (range 10)) => (1 2 3 4 5 6 7 8 9 10)`
 - `(reduce + (map inc (range 10))) => 55`

map and reduce

- `map`'s primary structure is
 - `(map function collection)`
- It returns a **new collection** in which `function` was applied to each member of the input `collection`
- Likewise `reduce`'s primary structure is
 - `(reduce function collection)`
- It returns a **single value** that is the result of repeatedly combining elements of the `collection` (in order) using the `function` (the function must support at least two arguments)
 - In the example on the previous slide, `reduce` first applied `+` to 1 and 2, it then applied `+` to 3 and 3, then `+` to 6 and 4, etc.

The Book's First Example: Imperative/Mutable

- The book starts with this program for inspiration
- ```
public int sum(int[] numbers) {
 • int accumulator = 0;
 • for (int n: numbers) {
 • accumulator += n;
 • }
 • return accumulator;
• }
```
- This is an imperative program to sum up an array of integers. `accumulator` is a **mutable** variable. We use an imperative `for` loop to tell the computer what to do

# The Book's First Example: Functional/Recursive

---

- `(defn recursive-sum [numbers]`
  - `(if (empty? numbers)`
    - `0`
    - `(+ (first numbers) (recursive-sum (rest numbers))))`
- This function is recursive in that it calls itself
  - It is functional in that there is no mutable state
    - At each point in the call stack, `numbers` is bound to different values
    - When `numbers` is empty, the recursion bottoms out and starts to unwrap, calculating as it goes
- This example introduces three new functions: `empty?`, `first`, and `rest`
  - `first` and `rest` are used to manipulate sequences (lists and vectors both can act as sequences)

# The Book's First Example: reduce

---

- As previously mentioned, functional programming will avoid recursion if it can; as such, the next version of this example is
- ```
(defn reduce-sum [numbers]  
  (reduce (fn [acc x] (+ acc x)) 0 numbers))
```
- This uses a version of reduce in which the initial value is also specified
- However, we don't need to define a function to add two numbers together, we already have one: +
- The final version of this function is thus
- ```
(defn sum [numbers] (reduce + numbers))
```
- Note: + automatically knows how to handle empty collections and collections consisting of just a single number (it uses its “identity” value of zero)

# The reward?

---

- How do we make our `sum` function concurrent?
- ```
(ns sum.core (:require [clojure.core.reducers :as r]))
```
- ```
(defn parallel-sum [numbers]
 (r/fold + numbers))
```
- This code pulls in a Clojure package called `reducers`. It aliases that package to the symbol `r` (so we don't have to type `reducers` all the time).
- The fold function is an implementation of `reduce` that (by default) breaks its input collection into groups of 512 elements each and performs the reduce calculation (in this case `+`) in parallel across all of the host machine's cores
  - ```
(time (sum numbers)) ; "Elapsed time: 91.193 msecs"
```
 - ```
(time (parallel-sum numbers)) ; "Elapsed time: 46.388 msecs"
```
- One call to a drop-in replacement of `reduce` and you're done!

# The Book's Second Example: Word Counts (I)

---

- The book's second example returns to the Word Counts example
  - i.e. count all of the words in the first 39,000 pages of Wikipedia articles

- Quick Intro to Maps (hash tables) in Clojure

```
(def counts {"apple" 2 "orange" 1})
(get counts "apple" 0) => 2
(get counts "banana" 0) => 0
(assoc counts "banana" 1) => {"apple" 2 "orange" 1 "banana" 1}
(assoc counts "apple" 3) => {"apple" 3 "orange" 1}
```

- Note that `assoc` returns a **NEW** map, the original map is **immutable**
  - If you really wanted to save the new map, you would need to bind it to a new symbol or rebind `counts` to the new value
    - `(def counts (assoc counts "banana" 1))`

# The Book's Second Example: Word Counts (II)

---

- We now know enough about maps to write a function that can count how many times we see a particular word in a sequence

```
(defn word-frequencies [words]
 (reduce
 (fn [counts word] (assoc counts word (inc (get counts word 0))))
 {} words))
```

- Take this daunting expression a bit at a time!
  - Define a function word-frequencies that takes a sequence called words
  - Call reduce on words passing in an empty map {} as the initial value
  - We reduce with an anonymous function with two parameters; It gets the current count associated with the current word, adds one to it, and sets that as the new count for that word
- Turns out that Clojure already has a function that does this: `frequencies`

# The Book's Second Example: Word Counts (III)

---

- Clojure has a concept known as a **partially applied function**
  - Our book is about to use it to perform word counting in parallel, so we need to understand it
- The basic concept is the following
  - A function takes  $n$  parameters
  - You are in a situation where you have  $k$  parameters for the function now (with  $k < n$ ) and you'll have the other  $(n-k)$  parameters later
  - You ask Clojure to create a new function that has your  $k$  parameters “wired in” as constants and takes as arguments the other  $(n-k)$  parameters later
  - You move forward with this new function and call it with the other parameters when the time comes

# The Book's Second Example: Word Counts (IV)

---

- Partially applied functions are perhaps easier to understand by examples
- Let's pretend we want to be able to add 5 to any set of integers
  - `(def add-five (partial + 5))`
- The form `(partial + 5)` says, “create a new function in which **5** has been hardwired in as `+`'s first argument”
- The new function `add-five` now acts just like `+` but it always has **5** as one of its inputs
  - `(add-five) => 5`
  - `(add-five 10) => 15`
  - `(add-five 10 10 10 10) => 45`

# The Book's Second Example: Word Counts (V)

---

- `partial` can be applied to any function
  - `(def add-five-to-everything (partial map add-five))`
- Here we bind the `add-five` function to `map`'s first parameter
  - With the resulting function, we just need to pass in the collection that `map` needs to operate on
  - `(add-five-to-everything [10 20 30 40 50 60 70 80 90])`
    - **returns** `(15 25 35 45 55 65 75 85 95)`

# The Book's Second Example: Word Counts (VI)

---

- We need to understand four more Clojure functions/concepts
  - `re-seq`: applies a regular expression to a string and produces a lazy sequence of all matches
  - `mapcat`: takes a sequence of sequences and produces a single sequence of all the subsequences concatenated
  - `merge-with`: a function to combine multiple maps into a single map with a rule as to how to combine duplicate map entries
  - `lazy sequences`: Clojure can work with large sequences abstractly, only creating those portions of the sequence that it needs

# re-seq

---

- re-seq is simple to understand
  - You give it a sequence and a pattern. It looks for matches of the pattern and produces a new sequence that contains each match
- `(defn get-numbers [text] (re-seq #"\d+" text))`
- Here we pass in a string and get back a sequence of all numbers found in that string
  - `(get-numbers "123 Boulder Ave 256 Dash Drive 5678 Pyramid Lane")`
  - **returns** `("123" "256" "5678")`

# mapcat

---

- You sometimes perform map operations that produce a sequence of sequences
- `(map get-numbers ["123B456", "789T101112", "131415G161718"])`
  - **returns** `(("123" "456") ("789" "101112") ("131415" "161718"))`
- Note that each element of the sequence is itself a sequence
- And sometimes you want that sequence of sequences to be “flattened” into a single sequence consisting of all the members of the subsequences
  - `(flatten (map get-numbers ["123B456", "789T101112", "131415G161718"]))`
    - **returns** `("123" "456" "789" "101112" "131415" "161718")`
- You can do this all in once step with `mapcat`
- `(mapcat get-numbers ["123B456", "789T101112", "131415G161718"])`
  - **returns** `("123" "456" "789" "101112" "131415" "161718")`

# merge-with

---

- `merge-with` allows you to combine multiple maps into a single map
  - It lets you specify what function is to be used to merge duplicate entries
- Given two maps
  - `(def counts1 {:ken 10 :max 20 :miles 10})`
  - `(def counts2 {:ken 40 :max 30 :lilja 50 :miles 40})`
- You can merge them and add their scores together with
  - `(merge-with + counts1 counts2)`
  - **returns** `{:lilja 50, :miles 50, :max 50, :ken 50}`

# Lazy Sequences (I)

---

- Clojure does what it can to avoid bringing an entire sequence into memory
  - It can instead pass around the “promise” of a sequence and then provide its elements when they are needed
- If you type `(range 0 100000000)` into the REPL and hit return
  - you may eventually see: `OutOfMemoryError Java heap space`
- Typing return means “display the result of evaluating this form”
  - it wants to display the sequence for you, which means it has to create it and then display it
- But, if you type `(def lots-of-numbers (range 0 100000000))` it returns instantly
  - That’s because the call to `range` is not evaluated until the elements of the sequence are needed

# Lazy Sequences (II)

---

- Lazy sequences work across any level of function calling
  - `(def lots-of-numbers-times-two (map (partial * 2) (range 0 10000000)))`
- Here it looks like we are saying
  - create a sequence with 10M members
  - Use the `map` function to multiply each of those numbers by 2
- But, the calculation is not performed until we actually ask for the result
  - `(take 10 lots-of-numbers) => (0 1 2 3 4 5 6 7 8 9)`
  - `(take 10 lots-of-numbers-times-two) => (0 2 4 6 8 10 12 14 16 18)`
- In both cases, only the first ten members of the sequence are generated and then operated on
  - This is efficient and fast!

# Lazy Sequences (III)

---

- You can even get to the end of the list without too much memory strain
  - `(take 10 (drop 9000000 lots-of-numbers-times-two))`
- This says skip past the first 9M numbers of the sequence, then show me the next ten; it tries to be efficient while doing this, garbage collecting those items of the sequence that are no longer needed (it still requires SOME memory)
  - If your JVM has a nice amount of memory, this operation is fast too
    - **Returns** `(18000000 18000002 18000004 18000006 18000008 18000010 18000012 18000014 18000016 18000018)`
- You just have to avoid asking for the ENTIRE sequence to be processed
  - If you do, then Clojure can't help it; it will bring the entire sequence into memory and then operate on it. You'll need to configure the JVM to have enough memory to handle the large sequence

# The new Word Count program

---

- The new Word Count program consists of three source files
  - `pages.clj`, `words.clj`, and `core.clj`
- In `pages.clj` is some functional XML parsing code that will make you lie in bed awake, unable to sleep at night
  - You can ignore it, it simply parses the XML file and gives us back the text of each Wikipedia article as a string via a function called `get-pages`
- `words.clj` defines the following function
  - ```
(defn get-words [text] (re-seq #"\\w+" text))
```
- As we just learned, `re-seq` will apply the regular expression to the string that represents the Wikipedia article and return each word in a sequence
 - That leaves the code in `core.clj` to handle the rest of the counting logic

Sequential Version

- To count all the words in a set of pages in a single thread, we use
 - ```
(defn count-words-sequential [pages]
 (frequencies (mapcat get-words pages)))
```
- This function
  - calls `get-words` on the passed in set of pages to generate a sequence of sequences containing the words for each page
  - and uses `mapcat` to ensure that we get a single (lazy) sequence of all such words
  - It then calls `frequencies` to produce a map that for each word tracks how many times it appears

# Sequential Version Performance

---

- To use it we call the following form
  - ```
(time (count (count-words-sequential (take 30000 (get-pages "enwiki.xml")))))
```
- I include a call to “count” to make Clojure actually perform the calculation
 - since otherwise with lazy sequences, it can decide not to do anything
 - plus the call to count allows me to see the output of the “time” function which otherwise gets lost when a map with 1.74M entries prints out!
 - Note: I switched to 30K pages because 39K pages was causing “out of memory” errors that I could not debug! :-)
- The sequential version of the program on 30K pages averages 71 seconds

Making it parallel: first attempt

- `(defn count-words-parallel [pages]`
 - `(reduce`
 - `(partial merge-with +)`
 - `(pmap #(frequencies (get-words %)) pages)))`
- Wow! Let's take that step by step
 - For each page, get its words, and calculate the frequencies
 - Supposedly do all of that in parallel with `pmap`
 - Then, reduce all of the maps into a single map using `merge-with`
 - Supposedly do that sequentially at the end
- The average running time is 46 seconds, only 35% faster
 - One reason: not all that concurrent, CPU usage was ~225%

Why is it slow (i.e. not as fast as we would like)?

- I said “supposedly” on the previous page
 - because lazy sequences actually alter the specified behavior
- Rather than performing all of that code in parallel
 - it was realizing the sequence, page by page, rather than all at once
 - Furthermore, it was creating one page, then merging it with the final map
 - and then creating the next page and merging it again
- This was similar to what we saw in Chapter 2 when our multiple consumer threads were all sharing a single `counts` map
 - and the program was slowed by contention around access to that map

Making it parallel: second attempt

- `(defn count-words [pages]`
 - `(reduce`
 - `(partial merge-with +)`
 - `(pmap count-words-seq (partition-all 100 pages))))`
- To fix this problem, we have to use the same approach we took in Chapter 2
 - We need to allow multiple counts to occur in parallel and merge into the final counts data structure only occasionally
- This version of `count-words`, uses `partition-all` to divide the 30K pages into 100 page chunks. `count-words-sequential` is used to count each of those 100 pages in parallel using `pmap`, THEN we merge into the final counts
 - With 100 page partitions, average run time 30.6 seconds with 400% CPU
 - With 500 page partitions, average run time 26.6 seconds with 500% CPU

Summary

- Today, we learned a lot about Clojure
 - its syntax, data structures, and functions
- We then examined how “simple” it is to transform single threaded programs to concurrent programs in the functional paradigm
 - Typically, we swap a single threaded version of a function with a concurrent version of that same function
 - reduce with fold; map with pmap
- Concurrency never comes for free however
 - The semantics of lazy sequences make taking advantage of full parallelization difficult to achieve
 - although without them, our program would have tried to load 30K wikipedia articles into memory!

Coming Up Next

- Lecture 12: Clojure Concurrency Constructs
- Lecture 13: User Stories, Part 5