CS 696 Functional Programming and Design
Fall Semester, 2015
Doc 12 Concurrency
Oct 6, 2015

Some Concurrency Background

Issues with Asynchronous Code

- Error Handing
- Read/Write Conflicts
- Communications between threads
    - Joins
    - Passing data back
- Callback Hell

Callback Hell

JavaScript problem core.async proposes to solve

Will use examples from Node.js

Node.js

Runs on Chrome's JavaScript runtime

Goal: fast, scalable networking applications

Event-driven non-blocking I/O
- So lightweight & efficient

Blocking I/O - Java

```
Path file = ...;
String fileContents = null;
try (InputStream in = Files.newInputStream(file);
    BufferedReader reader =
      new BufferedReader(new InputStreamReader(in))) {
    String line = null;
    while ((line = reader.readLine()) != null) {
        fileContents = fileContents + line;
    }
} catch (IOException x) {
    System.err.println(x);
}
return fileContents;
```

Blocking I/O - Clojure

```
(slurp “someFile.txt”)

(slurp “http://www.sdsu.edu”)
```

Non-Blocking I/O - Node.js

fs.readFile

- Reads a file asynchronously
- Need to provide function to process file contents

```
function processFooFile(error, fooFileContents) {
     if (error)
          throw error;
     Processes the file contents;
}

fs.readFile(‘filename.txt’, ‘utf-8’, processFooFile)
```

Reading Two Files - Node.js

```
function processFooFile(error, fooFileContents) {
     function processFoo&Bar(barError, barFileContents) {
    if (barError)
               throw barError;
    Process foo and bar contents here
     }

     if (error)
          throw error;
  }
     fs.readFile(‘bar.txt’, ‘utf-8’, processFoo&Bar);
}

fs.readFile(‘foo.txt’, ‘utf-8’, processFooFile)
```

Delay & Future

Delay

Suspends execution of code until delay is dereferenced

Caches result

Second time dereferenced returns cached result

Thread safe

```
(def wait (delay (println "do it now") (+ 1 2)))

 @wait                  prints "do it now", returns 3
 @wait                  returns 3
```

realized?

Returns true if a value has been produced for a promise, delay, future or lazy sequence.

```
(def wait (delay (println "do it now") (+ 1 2)))
```

```
(realized? wait)   false
@wait                  prints "do it now", returns 3
(realized? wait)   true
@wait                  returns 3
```

Example - Proxy for Expensive Operation

```
(defn fetch-page
  [url]
  {:url url
   :contents (delay (slurp url))})
```

```
(def result (fetch-page "http://www.eli.sdsu.edu/index.html"))

(:contents result)                   #<Delay@2fcc470c: :pending>

(realized? (:contents result))       false

@(:contents result)                  "<!DOCTYPE html>\n<html lang=\"en\">\n …"
```

@ and deref

```
@(:contents result)
```

```
(deref (:contents result))
```

They do the same thing

Future

Computes body on another thread

Use @ or deref to get answer

@, deref blocks until computation is done

```
(def long-calculation (future (apply + (range 1e8))))
@long-calculation
```

Future & Delay in ending program

When you end your program there will be a 1 minute delay if you used future

End your program with (shutdown-agents)

```
(def long-calculation (future (apply + (range 1e8))))

@long-calculation

(shutdown-agents)
```

(shutdown-agents) & REPL

(shutdown-agents) shuts down your REPL

deref with Timeout

```
(deref (future (Thread/sleep 5000) :done!)
       1000
       :impatient!)
  ;= :impatient!
```

Promise

Promise

one-time, single values pipe

```
(def p (promise))
(realized? p)                  false
(deliver p 42)                 #<core$promise$reify__1707@3f0ba812: 42>
(realized? p)                  true
@p                                 42
(deliver p 50)                 nil
@p                             42
```

Promise

Simple way to send data back from thread

References
agents

Agents

Uncoordinated

Asynchronous - run in separate thread

I/O & functions with side affects are safe in agents

Agents are STM-aware

- Agents in transactions are only run once

Agents

Agents hold data

You send functions to agents to process the data

Processing is done in separate thread

Sending work to an Agent

send

Sends to thread pool limited by cores on machine

send-off

Sends to unlimited thread

Send

```
(send a f & args)
```

Apply f to agent a with args

```
(apply f a args)
```

```
(def a (agent 500))
(send a range 1000)
@a
```

How does one know when Agent is Done

```
(await & agents)

(await-for timeout-ms & agents)
```

```
(def a (agent 50000))
(send a #(Thread/sleep %))
(await a)
@a
```

Exceptions in Agents

Agents are run on other thread

Exception in agents are not propagated back to main thread

agent-error

```
(def a (agent 1))

(send a inc)

@a                                                                       2

(agent-error a)                                                     nil

(send a (fn [_] (throw (Exception. "something is wrong"))))

@a                                                                       2

(agent-error a)            #<Exception java.lang.Exception: something is wrong>

(send a identity)          Exception
```

Agent Error Handlers

```
(def a (agent nil
              :error-mode :continue
              :error-handler (fn [the-agent exception]
                             (.println System/out (.getMessage exception)))))
```

Example use of Agents - logging changes

Watches are run on the current thread
I/O (logging) is slow

Use agent to do the logging

```
(defn log-reference
  [reference & writer-agents]
  (add-watch reference :log
           (fn [_ reference old new]
             (doseq [writer-agent writer-agents]
               (send-off writer-agent write new)))))
```

The Write & some Agents

```
 (defn write
     [^java.io.Writer w & content]
     (doseq [x (interpose " " content)]
       (.write w (str x)))
     (.write w "\n")
     (.flush w)
     w)


(def console (agent *out*))
(def character-log (agent (clojure.java.io/writer "character-states.log" :append true)))
```

```
(def cat 5)
(log-reference (var cat) console character-log)
(def cat 10)
```

Communicating Sequential Processes CSP

CSP

1978 - C. A. R. Hoare first descriped

Mathematical theory of concurrency

- Message passing & Channels

Used to specify & verify Concurrent systems

- T9000 Transputer

Influenced design of programming languages

- Occam
- Go

core.async

Added to Clojure 1.5

Provides independent threads of activity

- Communicating vai queue like channels

Supports

- Real threads & shared use of thread pools
- ClojureScript on JS engines (no threads)

Goals

- Simplify efficient server-side programs
- Simpler & more robust techniques for front-end ClojureScript programming

core.async Verses agents

Agents send functions to data

core.async sends data to functions

core.async

Not part of the standard library

```
:dependencies [[org.clojure/clojure "1.6.0"]
               [org.clojure/core.async "0.1.346.0-17112a-alpha"]]
```

For Examples

```
(ns basiclectures.basic-language.async-example
  (:require [clojure.core.async :as async]))
```

Channel

Communication link between producers and consumers

Channels can be

- Unbuffered
- Buffered

Types of Buffers

buffer

  blocks/parks when full

dropping-buffer

  While full drops items that are added

sliding-buffer

  While full drops oldest item when new item added

Producing a Channel

(chan)
(chan buf-or-n)

(chan 5)	channel with buffer of size 5
(chan (buffer 3))	channel with buffer of size 3
(chan (dropping-buffer 6))	
(chan (slidding-buffer 2))	

Reading/Writing Channels

(>!! channel value)

Writes value to channel

Blocks if buffer is full (unless buffer is sliding or drop)

(<!! channel)

Reads a value from channel

Blocks if nothing is available

Returns nil if channel is closed

Example

```
(def test-channel (async/chan 2))

(async/>!! test-channel "hello there")

(async/<!! test-channel)
```

Running in other Threads

futures

async/thread

go block

async/thread

```
(thread & body)
```

Runs body in separate thread

```
(async/thread (println "Hello"))
```

```
(def adder (async/thread (+ 1 2)))
(async/<!! adder)                         returns 3
```

```
(defn producer
  [channel name]

  (doseq [x [1 2 "end"]]
    (do
      (Thread/sleep 100)
      (println  name "producing " x)
      (async/>!! channel x)))
  (async/close! channel))
```

```
(defn consumer
  [channel]
  (let [input (async/<!! channel)]
    (println "input" input)
    (when input
      (recur channel))))
```

```
(let [channel (async/chan 7)]
  (println "Start")
  (async/thread (producer channel "a"))
  (async/thread (producer channel "b"))
  (async/thread (consumer channel)))
```

Issues

How to tell consumer we are done?

Producers use thread even when they are idle

Using Atom

```
(defn consumer
  [channel]
  (let [input (atom "start")]
    (while @input
      (do
        (reset! input (async/<!! channel))
        (println "consumming" @input)))))
```

go blocks

```
(go & body)
```

Executes body using thread in thread pool

When body blocks thread is released

When body unblocks run on a thread

ClojureScript

- Required to use channels
- Run on event loop

go blocks

```
(async/go (println “hello”))

(def adder (async/go (+ 1 2)))

(async/<!! adder)
```

go blocks

<!	use to read from channel instead of <!!
>!	use to write to channel instead of >!

```
(let [c (async/chan)]
  (async/go (>! c "hello"))
  (assert (= "hello" (async/<!! (async/go (<! c)))))
  (close! c))
```

>! verses >!!

```
(let [c (async/chan)]
  (async/go (>! c "hello")))
```

```
(defn hello
      [channel]
      (async/>!! channel “hello”))

(let [c (async/chan)]
  (async/go (hello c)))
```

Producer Example

```
(let [channel (async/chan 7)]
  (println "Start")
  (async/go (producer channel "a"))
  (async/go (producer channel "b"))
  (async/go (consumer channel)))
```

go blocks are lightweight

```
(let [n 1000
      cs (repeatedly n async/chan)
      begin (System/currentTimeMillis)]
  (doseq [c cs] (async/go (async/>! c "hi")))

  (dotimes [i n]
    (let [[v c] (async/alts!! cs)]
      (assert (= "hi" v))))
  (println "Read" n "msgs in" (- (System/currentTimeMillis) begin) "ms"))
```

alts!! & alts!

(alts! channels & {:as opts}

Takes value from one of the channels that have data

```
(let [c1 (async/chan)
      c2 (async/chan)]
  (async/thread (while true
      (let [[v ch] (async/alts! [c1 c2])]
        (println "Read" v "from" ch))))
  (async/go (async/>! c1 "hi"))
  (async/go (async/>! c2 "there")))
```

map, reduce, filter on Channels

```
(def simple-chan (async/chan 2))
(def inc-chan (async/map< inc simple-chan))

(async/>!! inc-chan 1)
(async/<!! inc-chan)                              returns 2
```

Rock Paper Scissors Example

```
(def MOVES [:rock :paper :scissors])
(def BEATS {:rock :scissors, :paper :rock, :scissors :paper})


(defn winner
  "Based on two moves, return the name of the winner."
  [[name1 move1] [name2 move2]]
  (cond
   (= move1 move2) "no one"
   (= move2 (BEATS move1)) name1
   :else name2))
```

Source http://tech.puredanger.com/2013/07/10/rps-core-async/

Report - Helper

```
(defn report
  "Report results of a match to the console."
  [[name1 move1] [name2 move2] winner]
  (println)
  (println name1 "throws" move1)
  (println name2 "throws" move2)
  (println winner "wins!"))
```

Player

```
(defn rand-player
  "Create a named player and return a channel to report moves."
  [name]
  (let [out (async/chan)]
    (async/go (while true (async/>! out [name (rand-nth MOVES)])))
    out))
```

Judging results

```
(defn judge
  "Given two channels on which players report moves, create and return an
   output channel to report the results of each match as [move1 move2 winner]."
  [p1 p2]
  (let [out (async/chan)]
    (async/go
     (while true
       (let [m1 (async/<! p1)
             m2 (async/<! p2)]
         (async/>! out [m1 m2 (winner m1 m2)]))))
    out))
```

Playing single game

```
(defn init
  "Create 2 players (by default Alice and Bob) and return an output channel
of match results."
  ([] (init "Alice" "Bob"))
  ([n1 n2] (judge (rand-player n1) (rand-player n2))))

(defn play
  "Play by taking a match reporting channel and reporting the results of the latest
match."
  [out-chan]
  (apply report (async/<!! out-chan)))

  (play (init))
```

Playing Multiple Games

```
(defn play-many
  "Play n matches from out-chan and report a summary of the results."
  [out-chan n]
  (loop [remaining n
         results {}]
    (if (zero? remaining)
      results
      (let [[m1 m2 winner] (async/<!! out-chan)]
        (recur (dec remaining)
               (merge-with + results {winner 1}))))))
```

Multiple Games

```
(play-many game 10000)
```

```
{"Alice" 3323, "Bob" 3326, "no one" 3351}

"Elapsed time: 650.433 msecs"
```

rock paper scissors lizard spock

Try modifying code to play “rock paper scissors lizard spock”