# Virtual Threads in Clojure

## Implications and practices

**Eric Normand - May 7, 2024**

OS threads are limited
(bottleneck)

# Little’s law

time to process request

$$L = r\,W$$

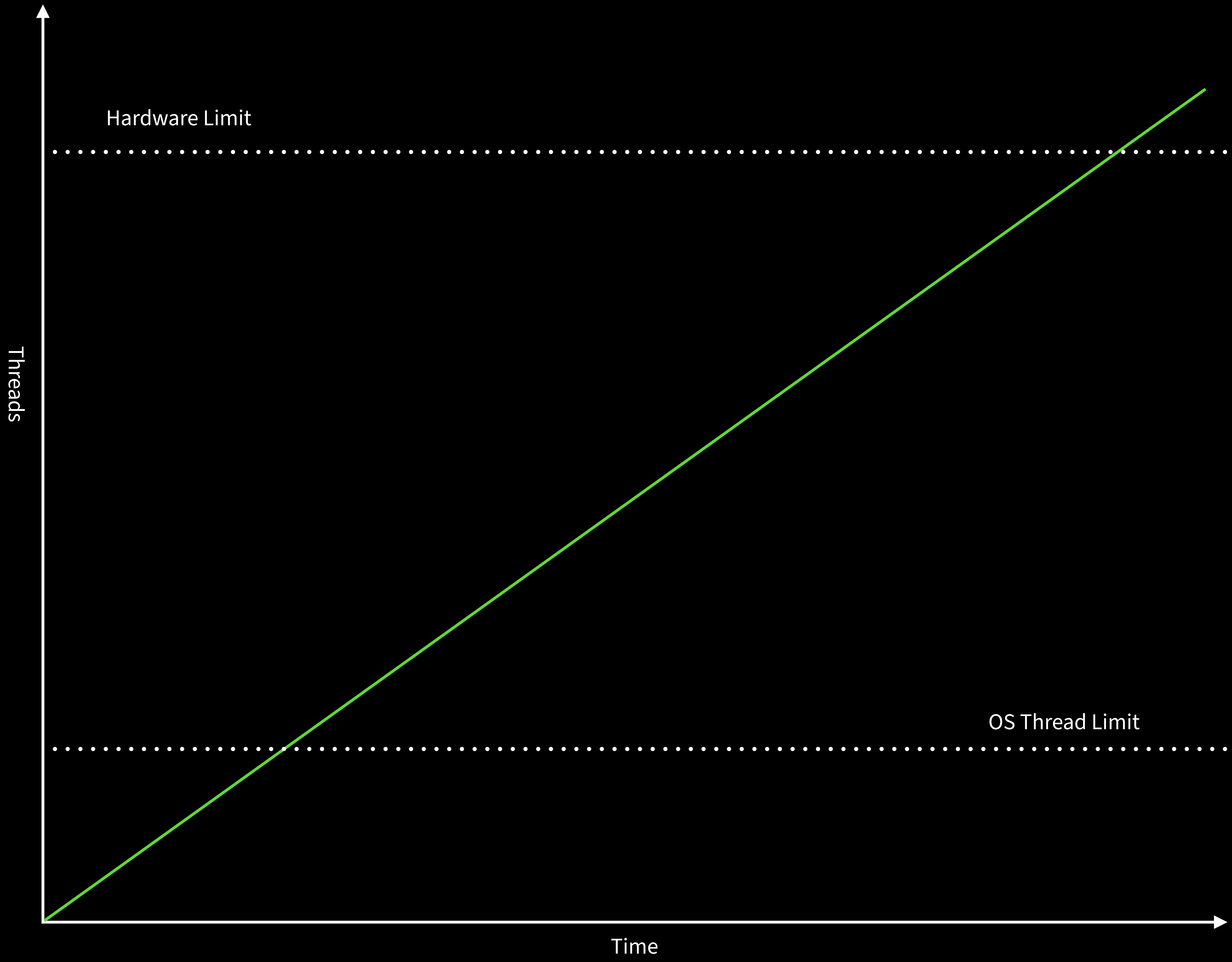
concurrent requests

requests per second

# $L = r\,W$

## Thread-per-request HTTP server

- Year 1
  - $r$ = 200 requests per second
  - $W$ = 50 ms to process each request
  - $L$ = 200/s x 50ms = 10 requests => 10 threads
- Year 2
  - $r$ = 2000 requests per second
  - $W$ = 50 ms to process each request
  - $L$ = 2000/s x 50ms = 100 requests => 100 threads


Hardware Limit
OS Thread Limit
Threads
Time

# Async programming

## One possible solution

callbacks, core.async, Promesa, interceptors, ring async

- Benefits
  - Lightweight
  - Garbage-collectable

- Costs
  - Callback hell
    - Stacktraces!
    - Exceptions!
  - Can’t use existing libraries
  - Can’t use existing tooling

# Virtual Threads

## Basically all the benefits of async AND threads

threads implemented in the JVM, run on an OS thread pool

- Benefits
  - Lightweight
  - Garbage-collectable
  - Stacktraces
  - Exceptions
  - Existing libraries
  - Existing tooling (debuggers, profilers, etc.)

- Costs
  - Bottleneck moves elsewhere
  - Limitations
    - CPU-bound
    - Synchronized

# Brass tacks

- JDK 21 — LTS — adoptium.net
- Instance of `java.lang.Thread`
- Recommendation: Use one virtual thread per task
    - Example: One virtual thread per HTTP request
- Don’t pool them — let them run, end, and be garbage collected

# Things you oughtn’t to do

- CPU-bound computation
  - Hot-loops
  - atoms? refs?
  - Solutions:
    - `Thread.sleep()`
    - `Thread.yield()`
    - not using atoms?

- `synchronized` keyword
  - synchronized blocks and methods
  - `(locking …)` macro
  - Solutions:
    - `j.u.c.locks.ReentrantLock`

# Things you can do

- Blocking I/O
- Blocking primitives
  - Locks
  - Queues
  - Futures
  - core.async blocking operations `<!!`, `>!!`, etc.
- `Thread.sleep()` and `.yield()`

# Creating virtual threads

## 3 ways

```
j.u.c.Executors/newVirtualThreadPerTaskExecutor
```

```clojure
(defonce executor (Executors/newVirtualThreadPerTaskExecutor))

;; call .submit method with a 0-argument function
(def f (.submit executor (fn [] 4)))

(type f) ;; .submit returns a future

;; get the value with deref or the .get method
;; will block until the value is ready
@f
(.get f)
```

# Creating virtual threads

## 3 ways

```
java.lang.Thread/startVirtualThread
```

```clojure
(Thread/startVirtualThread #(println "Hello"))
```

# Creating virtual threads

## 3 ways

`java.lang.Thread/ofVirtual` Builder

```
(-> (Thread/ofVirtual) (.name "My Tread") (.start #(println “Wow")))
```

```
(-> (Thread/ofVirtual) (.unstarted #(println “Wow")))
```

# Sharing state without atoms or refs

## Single writer

```
(defonce keep-going? (atom true))
(defonce executor (Executors/newVirtualThreadPerTaskExecutor))

(dotimes [n 10]
  ;; loop with a sleep, so it's fine
  (.submit executor (fn []
                      (while @keep-going?
                        (println "Still alive!")
                        (Thread/sleep 1000)))))

;; signal to stop threads after 25 seconds
(.submit executor (fn []
                    (Thread/sleep 25000)
                    (reset! keep-going? false)))
```

# Sharing state without atoms or refs

## java.util.concurrent Collections

```
(import '(java.util.concurrent Executors ConcurrentHashMap CountDownLatch))
(defonce executor (Executors/newVirtualThreadPerTaskExecutor))

(defn fetch-urls [urls]
  (let [results (ConcurrentHashMap.)
        latch (CountDownLatch. (count urls))]
    (doseq [url urls]
      (.submit executor (fn []
                          (.put results url (slurp url))
                          (.countDown latch))))
    (.submit executor (fn []
                        (.await latch)
                        (into {} results)))))

@(fetch-urls ["http://example.com/1", "http://example.com/2", "http://example.com/3"])
```

# Sharing state without atoms or refs

## Not sharing state???

```clojure
(defonce executor (Executors/newVirtualThreadPerTaskExecutor))

(defn fetch-urls [urls]
  (.submit executor (fn []
                      (let [futures (doall
                                      (map (fn [url]
                                             (.submit executor #(vector url (slurp url))))
                                           urls)))]
                        (into {} (map deref) futures)))))

@(fetch-urls ["http://example.com/1", "http://example.com/2", "http://example.com/3"])
```

# Communication and coordination

- core.async
- Promesa
- Manifold
- `java.util.concurrent`
  - `CountdownLatch`
  - `ArrayBlockingQueue`
  - `Semaphore`
  - etc.

# What’s coming next?

## 2 related projects

- Structured Concurrency
    - Represent hierarchical tasks
    - Fan-out, fan-in
- Scoped Values
    - Immutable values scoped to a thread and its subthreads