The State of ~~Fibers~~ for the JVM
~~Lightweight~~ Threads
Virtual

and ...

Why all your JVM coroutines are cool, but broken,
(Kilim threads, Quasar fibers, Kotlin coroutines, etc.)
and Project Loom is gonna fix that!

Volkan Yazıcı
https://vlkan.com
@yazicivo

2019-12-10
~~2018-08-25~~
~~2018-03-08~~

Agenda

1) Motivation
2) History
3) Continuations
4) Processes, threads, and fibers
5) Project Loom
6) Structured concurrency
7) Scoped variables

Motivation: I/O

Quoting from Gor Nishanov's 2016 "LLVM Coroutines: Bringing resumable functions to LLVM" talk.

Without I/O, you don't exist!

- println()
- file access
- network socket access
- database access
- etc.

In 1958, ...

Quoting from Gor Nishanov's 2016 "LLVM Coroutines: Bringing resumable functions to LLVM" talk.

The legend of Melvin Conway

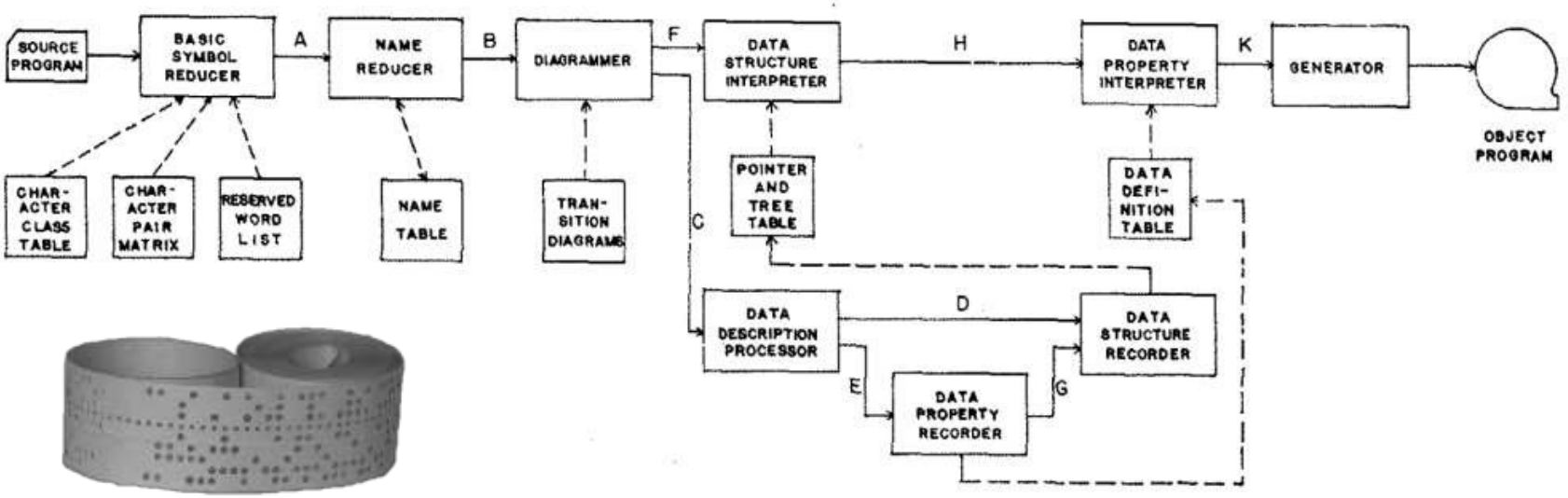


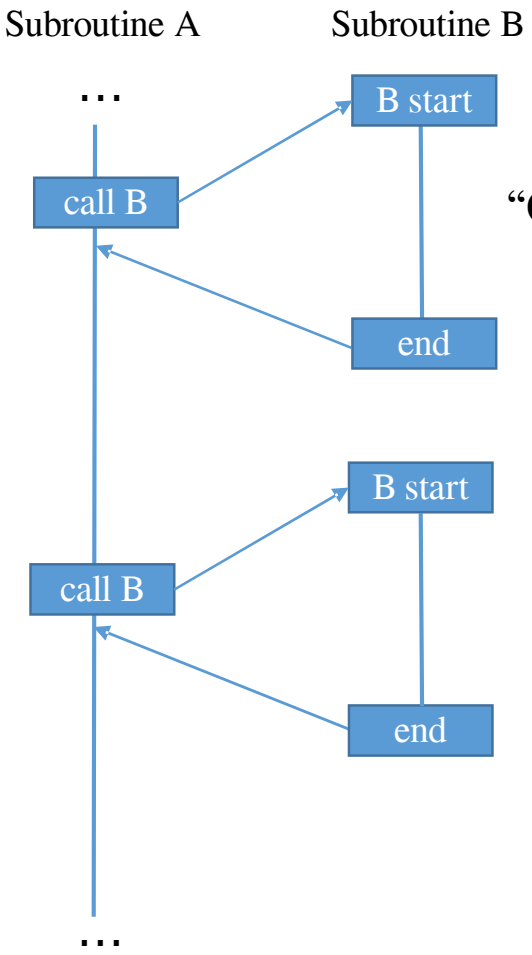
FIG. 4. COBOL Compiler Organization

Communications of the ACM

Volume 6 / Number 7 / July, 1963

Quoting from Gor Nishanov's 2016 "LLVM Coroutines: Bringing resumable functions to LLVM" talk.

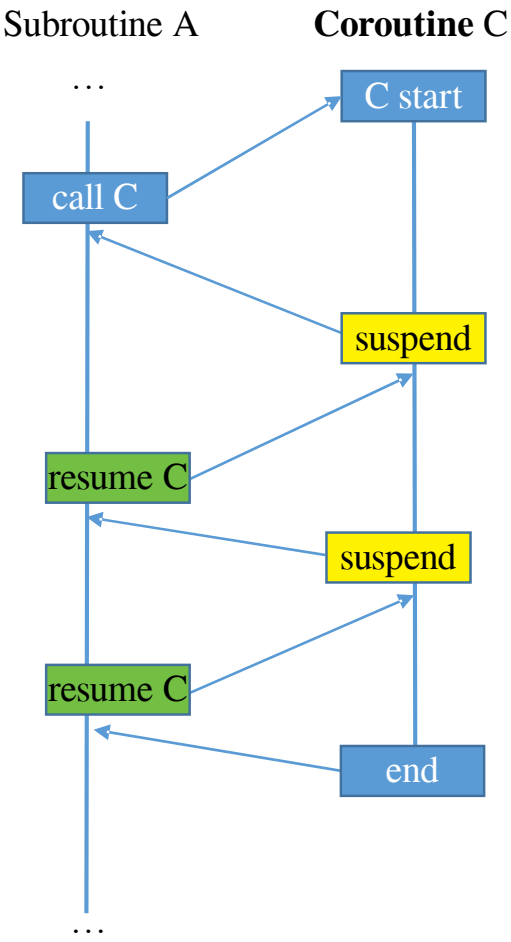
Subroutine ⊂ Coroutine



"Coroutines" – Melvin Conway, 1958

"Generalization of subroutine" – Donald Knuth, 1968

	subroutines	**coroutines**
call	allocate frame, pass params	allocate frame, pass params
return	free frame, return result	free frame, return result
suspend	No	Yes
resume	No	Yes



Quoting from Gor Nishanov's 2016 "LLVM Coroutines: Bringing resumable functions to LLVM" talk.

Where did all the coroutines go?

Algol-60

… introduced **code blocks** and the begin and end pairs for delimiting them. ALGOL 60 was the first language implementing **nested function definitions** with **lexical scope**.
– Wikipedia “ALGOL 60”

Quoting from Gor Nishanov's 2016 "LLVM Coroutines: Bringing resumable functions to LLVM" talk.

One "continuation" to rule them all...

"Continuation Sandwich"

- The earliest description by Adriaan van Wijngaarden in 1964.
- "… a data structure that represents the computational process at a given point in the process's execution; ..." – Wikipedia "Continuation"
- "Say you're in the kitchen in front of the refrigerator, thinking about a sandwich. You take a continuation right there and stick it in your pocket. Then you get some turkey and bread out of the refrigerator and make yourself a sandwich, which is now sitting on the counter. You invoke the continuation in your pocket, and you find yourself standing in front of the refrigerator again, thinking about a sandwich. But fortunately, there's a sandwich on the counter, and all the materials used to make it are gone. So you eat it." – Luke Palmer, 2004

A glimpse of "continuation"

```
(define the-continuation #f)

(define (test)
  (let ((i 0))
    ; call/cc calls its first function argument, passing
    ; a continuation variable representing this point in
    ; the program as the argument to that function.
    (call/cc (lambda (k) (set! the-continuation k)))
    ; The next time the-continuation is called, we start here.
    (set! i (+ i 1))
    i))
```

```
> (test)
1
> (the-continuation)
2
> (the-continuation)
3
> ; stores the current continuation (which will print 4 next) away
> (define another-continuation the-continuation)
> (test) ; resets the-continuation
1
> (the-continuation)
2
> (another-continuation) ; uses the previously stored continuation
4
```

See Oleg Kiselyo's An argument against call/cc and Undelimited continuations are not functions articles.

A glimpse of "delimited continuation"

Unlike regular continuations, delimited continuations return a value, and thus may be reused and composed.

```
; The reset delimits the continuation that shift captures (named by k in this example).
; The use of shift will bind k to the continuation (+ 1 []),
; where [] represents the part of the computation that is to be filled with a value.
(* 2 (reset (+ 1 (shift k (k 5)))))
```

```
(reset (* 2 (shift k (k (k 4)))))
; invokes (k 4) first (which returns 8),
; and then (k 8) (which returns 16).
; At this point, the shift expression has terminated,
; and the rest of the reset expression is discarded.
; Therefore, the final result is 16.
```

What is so important about continuations?

Using continuations you can implement

- longjmp (C)
- exceptions (C++, Java, etc.)
- generators (Icon, Python, etc.)
- backtracking (Prolog, etc.)
- and… guess what else?

continuation + scheduler = ?

How do we compose I/O?

(without changing neither the language, nor the VM byte code)

Blocking calls

```
public SearchResponse search(SearchRequest request) {

    // Check caches.
    SearchResponse cachedResponse = cache.find(reqest);
    if (cachedResponse != null) {
        return cachedResponse;
    }

    // Check redirects.
    SearchResponse redirectedResponse = redirectSer
    if (redirectedResponse != null) {
        return redirectedResponse;
    }

    // Perform plain search enriched with suggestic
    SearchRequest enrichedRequest = suggestionServi
    return plainSearch(enrichedRequest);

}
```

Callbacks (non-blocking)

```
public void search(
        SearchRequest request,
        Consumer<SearchResponse> callback) {

    // Check caches.
    return cache.find(request, cachedResponse -> {
        if (cachedResponse != null) {
            callback.accept(cachedResponse);
        } else {

            // Check redirects.
            redirectService.find(request, redirectedResponse -> {
                if (redirectedResponse != null) {
                    return callback.accept(redirectedResponse);
                } else {

                    // Perform plain search enriched with suggestions.
                    suggestionService.enrich(request, enrichedRequest -> {
                        plainSearch(enrichedRequest, searchResponse -> {
                            callback.accept(searchResponse);
                        });
                    });

                }
            })
        }
    });
}
```

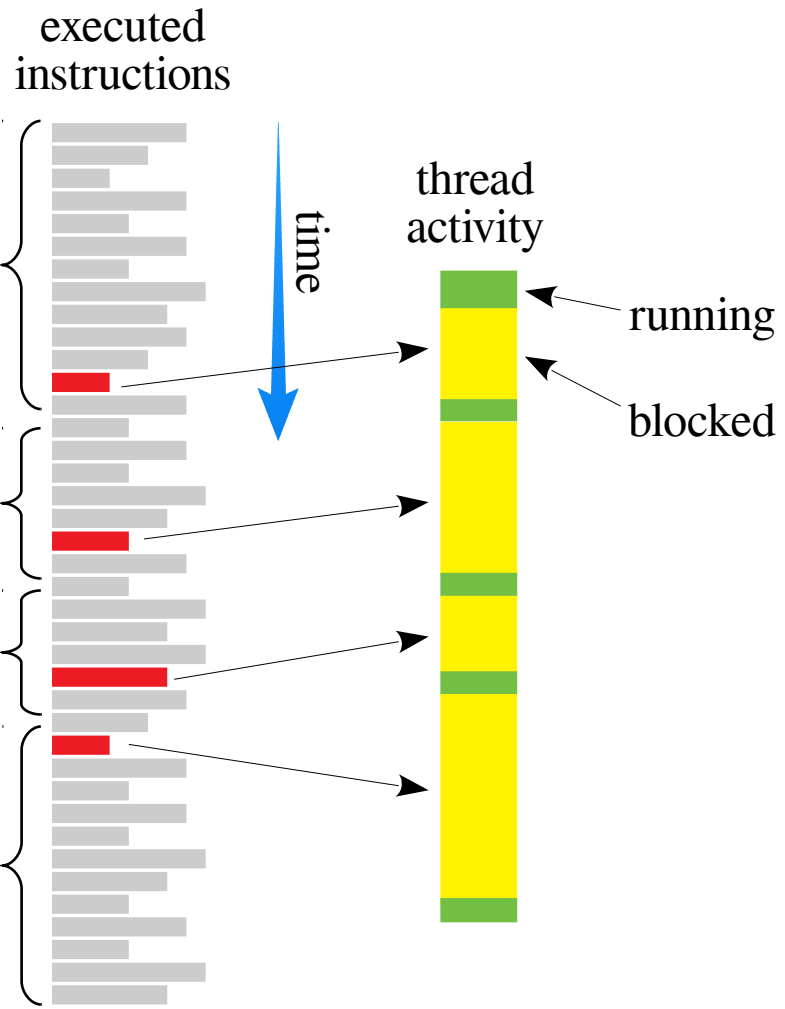
Reactor (non-blocking)

```
public Mono<SearchResponse> search(SearchRequest request) {
    return Flux
            .concat(cache.find(request),
                    redirectService.find(request),
                    suggestionService
                            .find(request)
                            .flatMap(this::plainSearch))
            .take(1)
            .singleOrEmpty();
}
```

Blocking what?

Blocking calls

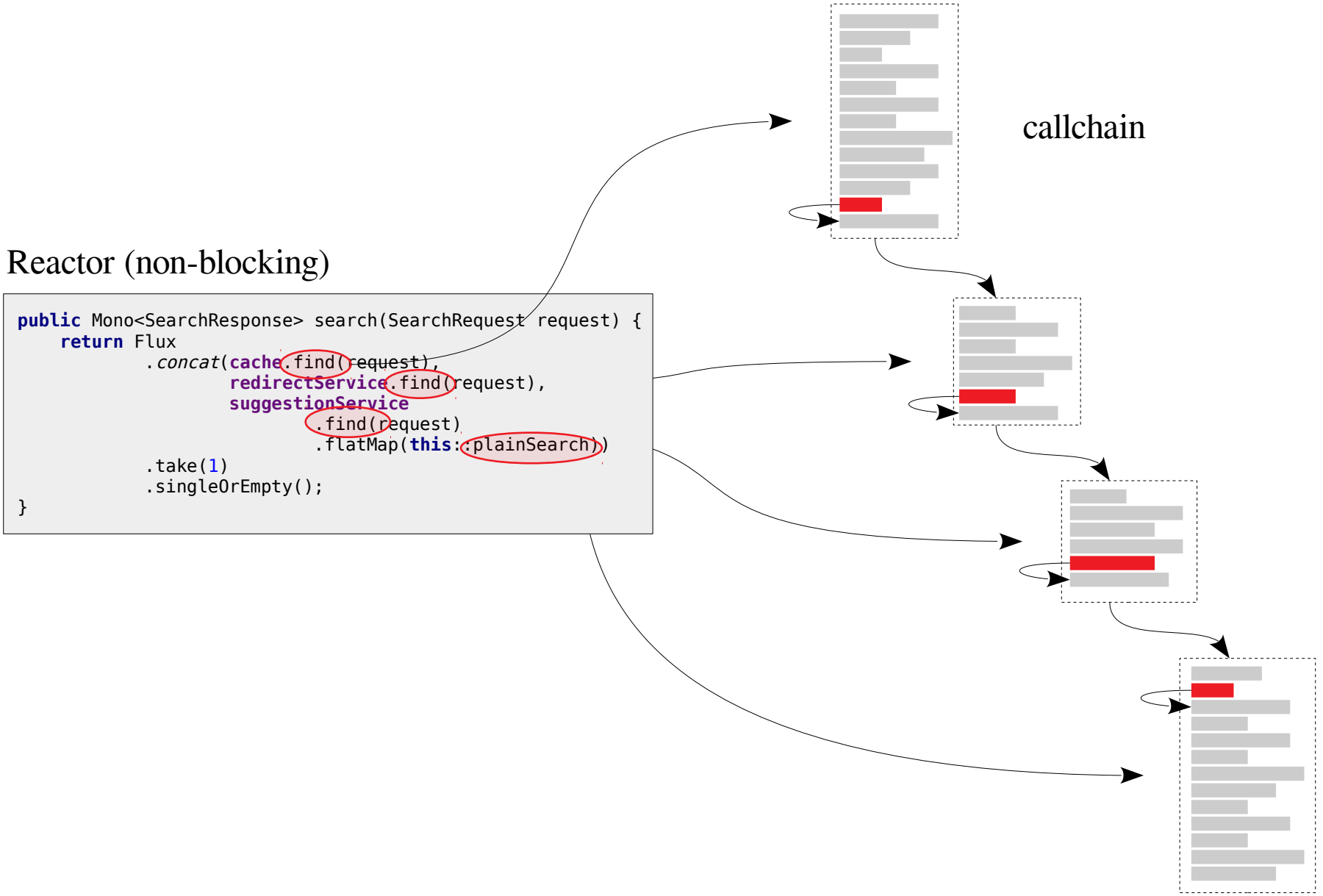
```
public SearchResponse search(SearchRequest request) {

    // Check caches.
    SearchResponse cachedResponse = cache.find(reqest);
    if (cachedResponse != null) {
        return cachedResponse;
    }

    // Check redirects.
    SearchResponse redirectedResponse = redirectService.find(request);
    if (redirectedResponse != null) {
        return redirectedResponse;
    }

    // Perform plain search enriched with suggestions.
    SearchRequest enrichedRequest = suggestionService.enrich(request);
    return plainSearch(enrichedRequest);

}
```



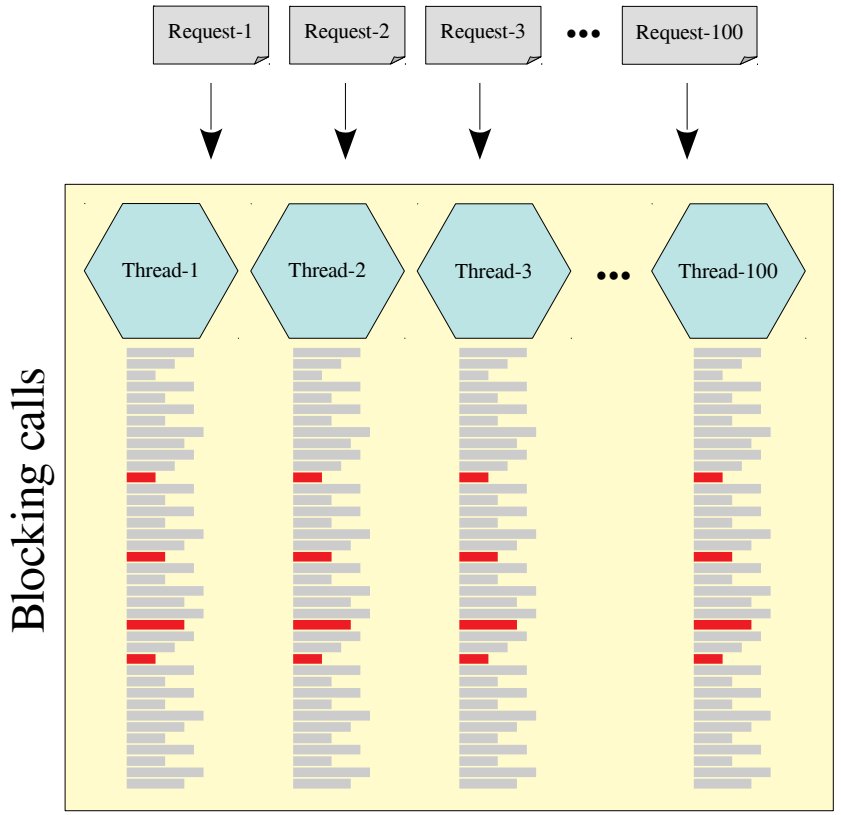
How does asynchronous I/O get composed?

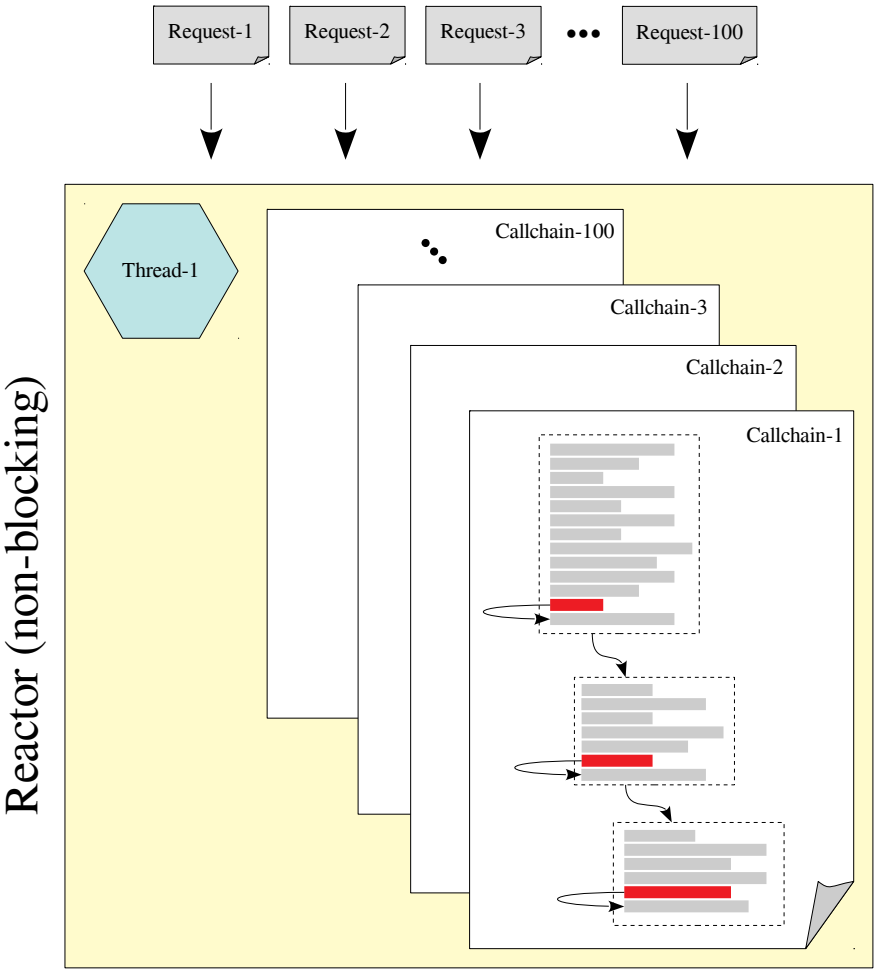
Reactor (non-blocking)

```
public Mono<SearchResponse> search(SearchRequest request) {
    return Flux
            .concat(cache.find(request),
                    redirectService.find(request),
                    suggestionService
                            .find(request)
                            .flatMap(this::plainSearch))
            .take(1)
            .singleOrEmpty();
}
```



Why all this reactive hassle?

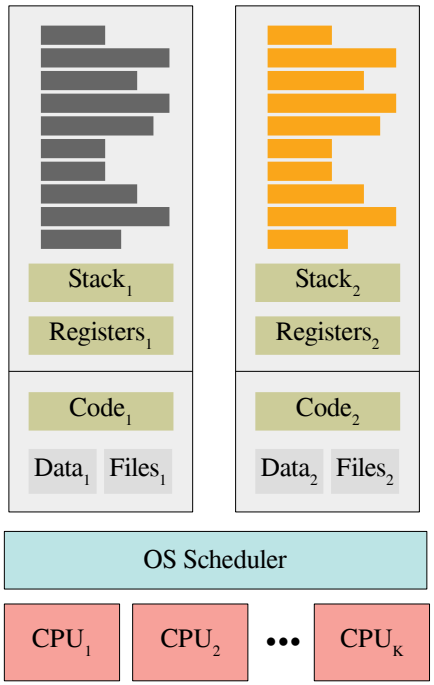




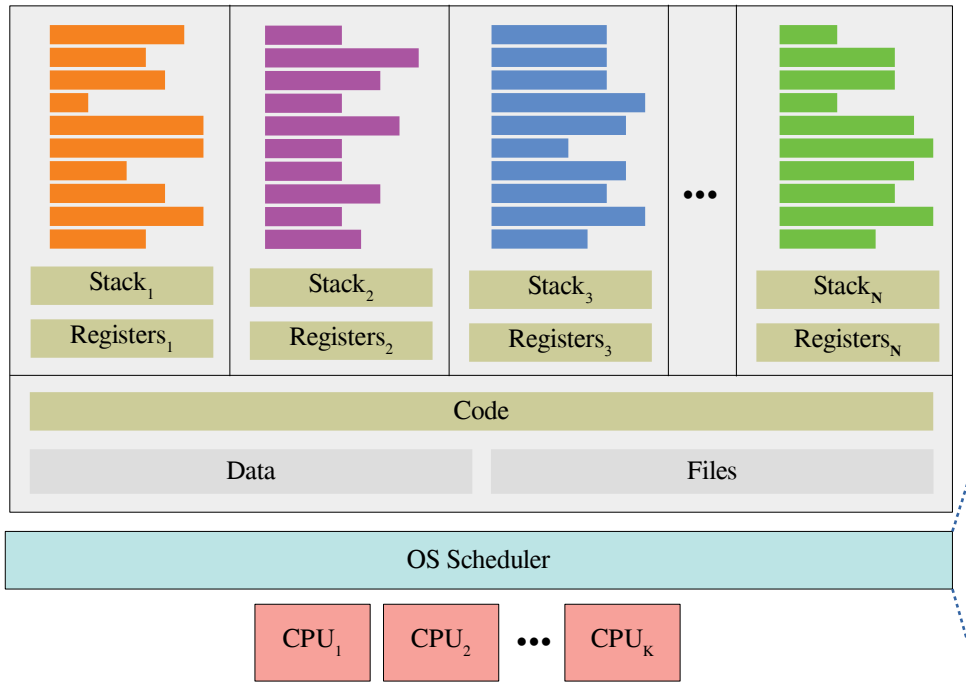
If **spawning** and **context switching** costs of threads would be equal to the ones in callchains, would you still favor the latter?

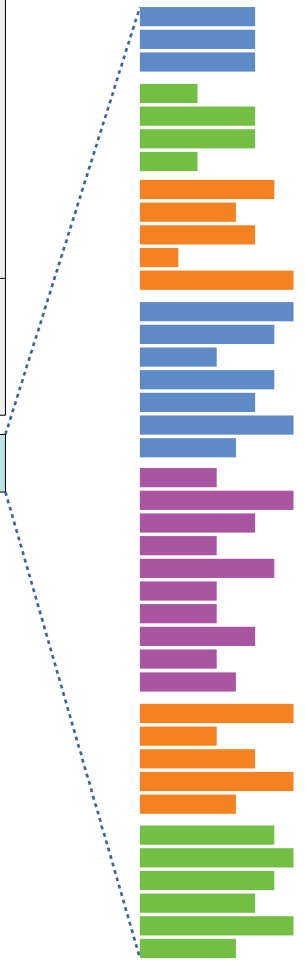
Process-vs-thread

(single-threaded)
OS Process



(multi-threaded)
OS Process





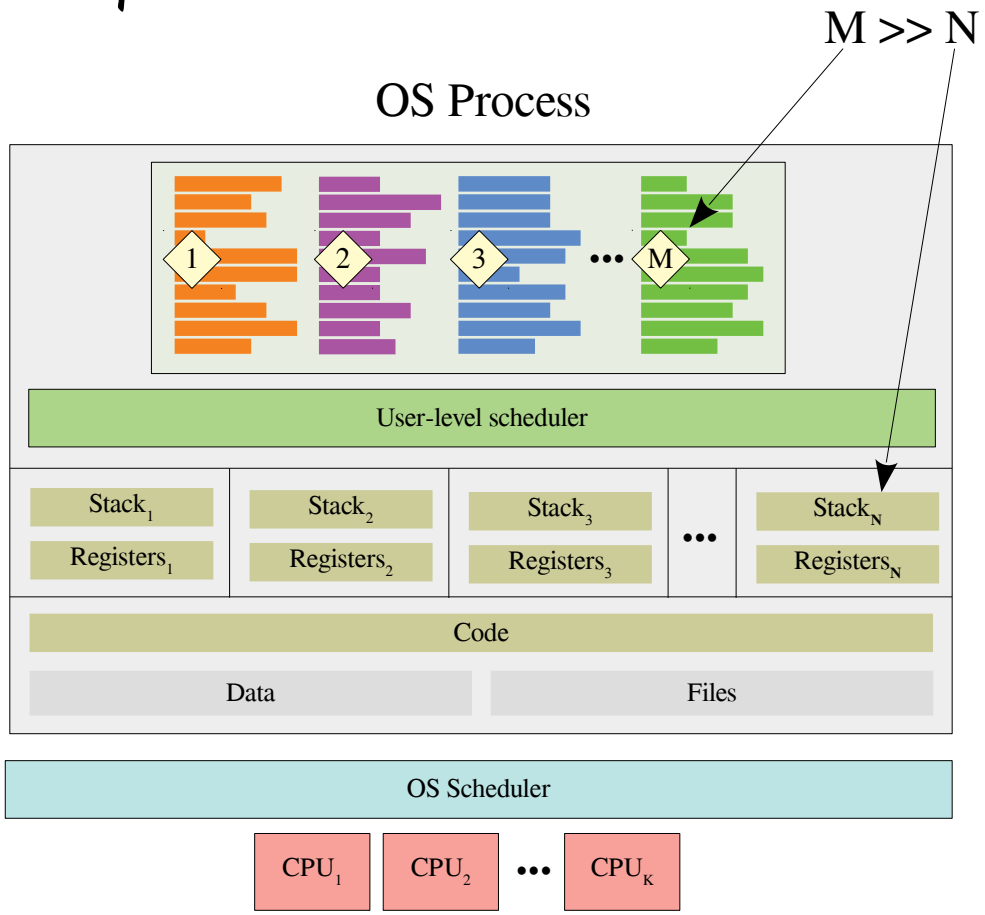
Both processes and threads denote a **continuation**: a sequence of instructions that can be ***suspend***ed and ***resume***d.

process = continuation + scheduler

Cheaper!

thread = continuation + scheduler

What is a fiber?



Both processes and threads denote a **continuation**: a sequence of instructions that can be ***suspend***ed and ***resume***d.

process = continuation + scheduler **kernel**

Cheaper!

thread = continuation + scheduler **kernel**

Cheaper!

fiber = continuation + scheduler **user**

both share memory

Does JVM support fibers?

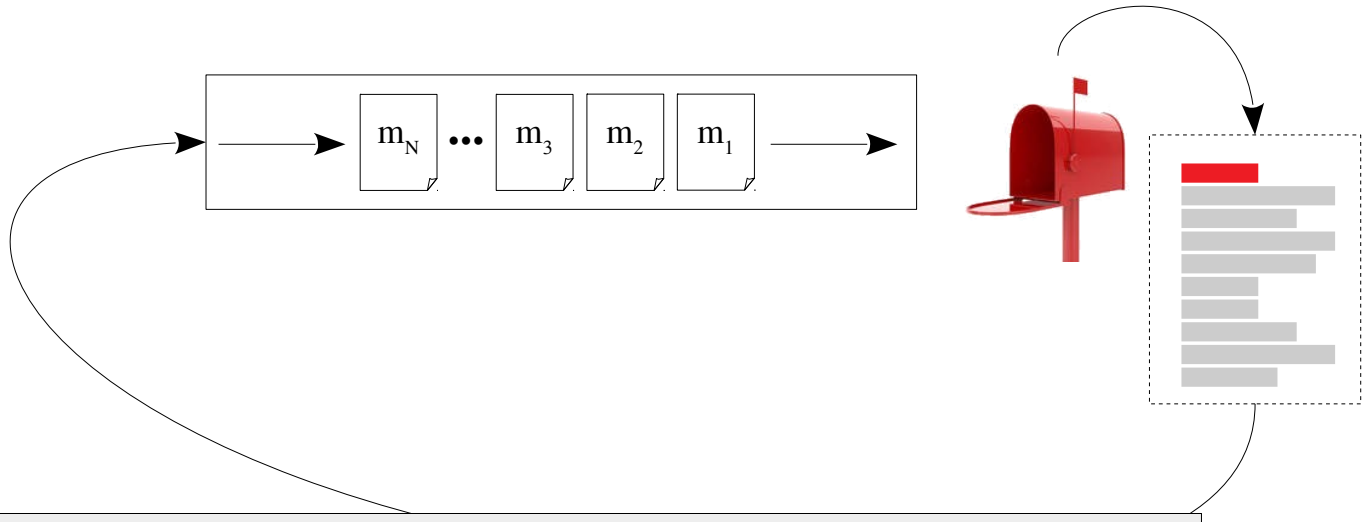
in essence, “continuations”

- Yes, it can, but it (natively) doesn’t.
- Lisp(call/cc), BEAM(Erlang VM), Haskell, Go, JavaScript(async, await) (natively) do.
- Quasar, Kilim, etc. provides continuations and fibers for JVM.
  - What if someone calls **Thread.sleep**()?
  - What if a coroutine calls a non-coroutine method?
    What if that non-coroutine method is blocking?
- What about Kotlin coroutines?

What about actors(Erlang, Akka) / channels(Go, Kotlin)?

```
class MyActor extends Actor {
  val log = Logging(context.system, this)

  def receive = {
    case "test" => log.info("received test")
    case _      => log.info("received unknown message")
  }
}
```

```
ch <- v    // Send v to channel ch.
v := <-ch  // Receive from ch, and
           // assign value to v.
```



fiber { BlockingQueue<V>(*bufferSize*) } = **Channel**<V>(*bufferSize*)

A glimpse of Quasar & Kilim

Quasar

```
new Fiber<V>() {
  @Override
  protected V run() throws SuspendExecution, InterruptedException {
        // your code
    }
}.start();
```

Kilim

```
new kilim.Task() {
    public void execute() throws Pausable, Exception {
        box.get();
        Task.sleep(1000);
        try {
            String result = handler.handle(target,br,req,resp);
            if (result != null) resp.getOutputStream().print(result);
        }
        catch (Exception ex) { resp.sendError(500,"the server encountered an error"); }
        br.setHandled(true);
        async.complete();
    }
}.start();
```

So what is the problem?

Asynchronous task composition

Callbacks

- Language-agnostic
- Difficult to write
  (callbackhell.com)
- Difficult to debug & profile
- Misses batteries
- Viral

Reactive Streams

- Language-agnostic
- Includes batteries
- Widely supported
- Need to learn a new language
- Difficult to write & get right
- Difficult to debug & profile
- Difficult to optimize (by compiler)
- Viral

Fibers

- Just works
- Language-specific
- Requires explicit marking
- Misses batteries

Reflective calls are always considered suspendable. This is because the target method is computed at runtime, so there's no general way of telling if it's going to call a suspendable method or not before execution.

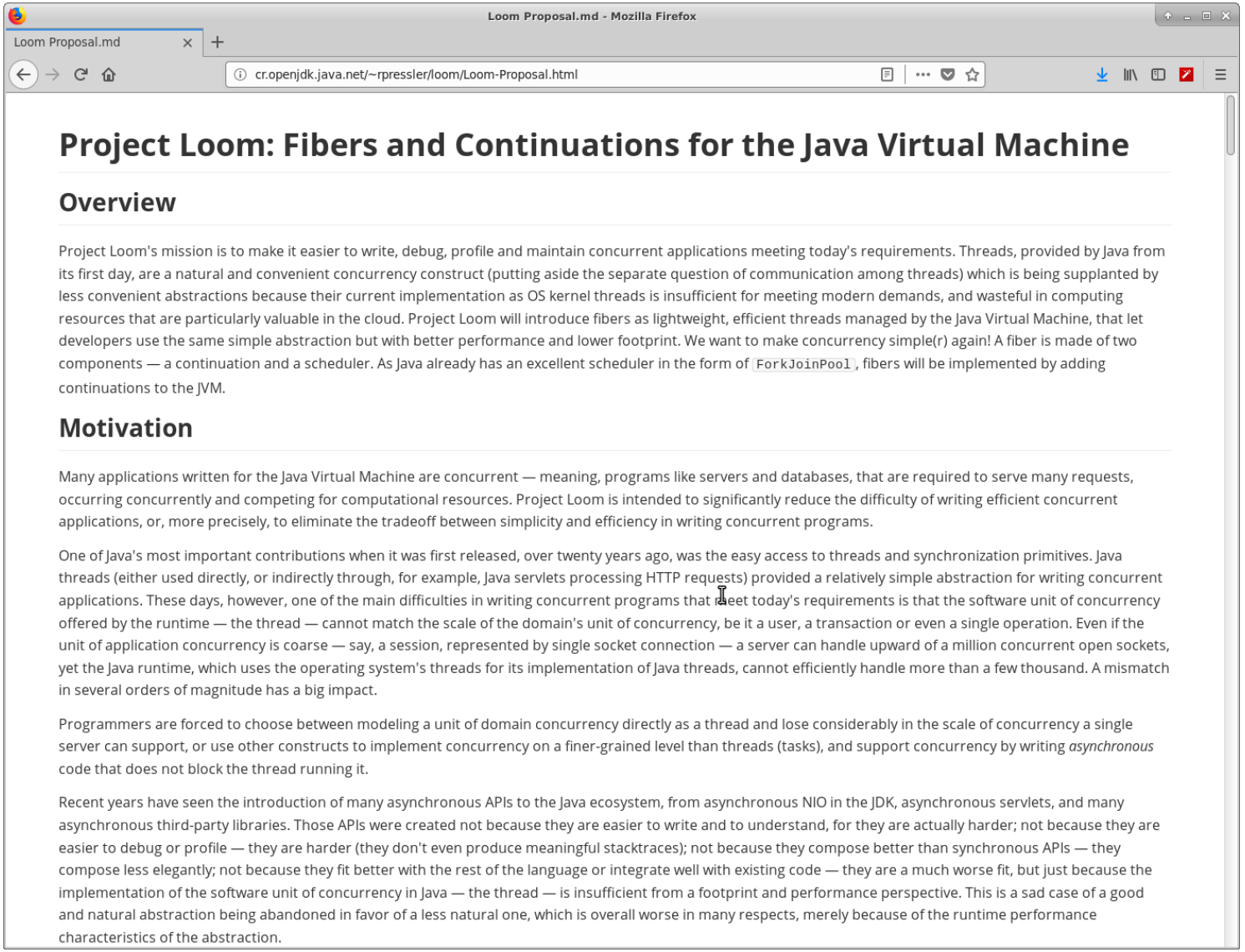
Java 8 lambdas too are always considered suspendable. This is because they can't declare checked exceptions, they are ultimately linked (via invokedynamic) to synthethic static methods that can't be annotated and it is difficult to tell at instrumentation time if lambdas implement a suspendable interface.

Quasar will reject with an error any attempt to mark special methods (that is, constructors and class initializers) as suspendable. This is because suspending in an initializer could expose objects or classes before they're fully initialized and this is an error-prone, difficult-to-troubleshoot situation that can always (and must) be avoided.

Enter Loom...

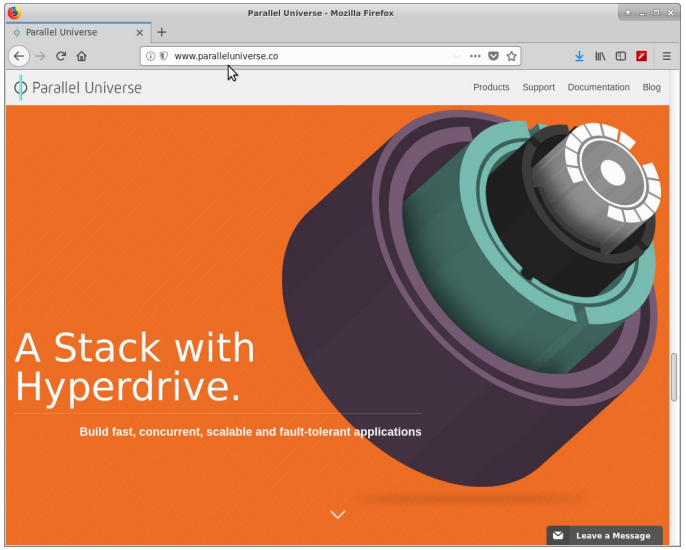
The proposal

Project Loom proposal
by Ron Pressler



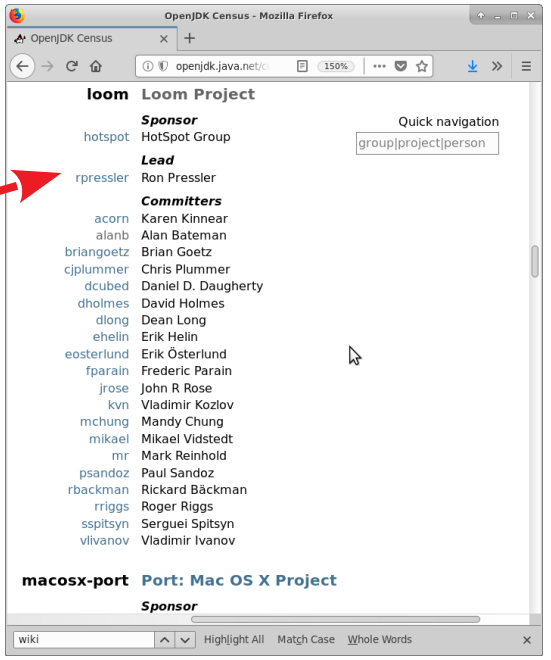
http://cr.openjdk.java.net/~rpressler/loom/Loom-Proposal.html

Who is Ron Pressler anyway?



http://www.paralleluniverse.co/

Lead at Project Loom (since 2017)



http://openjdk.java.net/census#loom

In 2012, founded Parallel Universe with the following F/OSS product line:

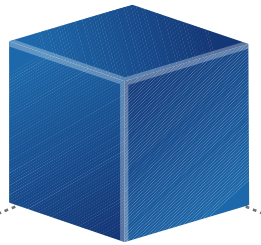
- **Quasar:** lightweight threads (fibers) for the JVM
- **Comsat:** fiber-aware impl's of servlets, JAX-RS/Spring REST services, HTTP clients and JDBC
- **SpaceBase:** in-memory spatial and geo-spatial database
- **Galaxy:** distributed in-memory data grid that horizontally scales

Proposal highlights

- One of Java's most important contributions when it was first released, over twenty years ago, was the easy access to threads and synchronization primitives.
- … today's requirements is that the software unit of concurrency offered by the runtime — the thread — cannot match the scale of the domain's unit of concurrency, …
- … asynchronous APIs ... were created
  - not because they are easier to write and to understand
    - **for they are actually harder**
  - not because they are easier to debug or profile
    - **they are harder (they don't even produce meaningful stacktraces)**
  - not because they compose better than synchronous APIs
    - **they compose less elegantly**
  - not because they fit better with the rest of the language or integrate well with existing code
    - **they are a much worse fit**
- but **just because the implementation of the software unit of concurrency in Java — the thread — is insufficient from a footprint and performance perspective.**

The key project deliverable

Lightweight threads

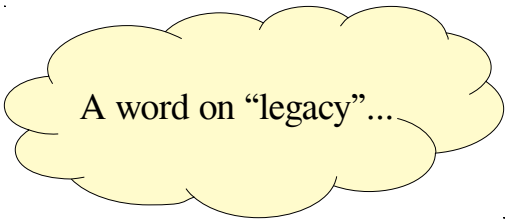


Asymmetric
one-shot (non-reentrant)
stackful
multi-prompt
delimited
continuations.

See Ron Pressler's detailed post in the loom-dev mailing list.

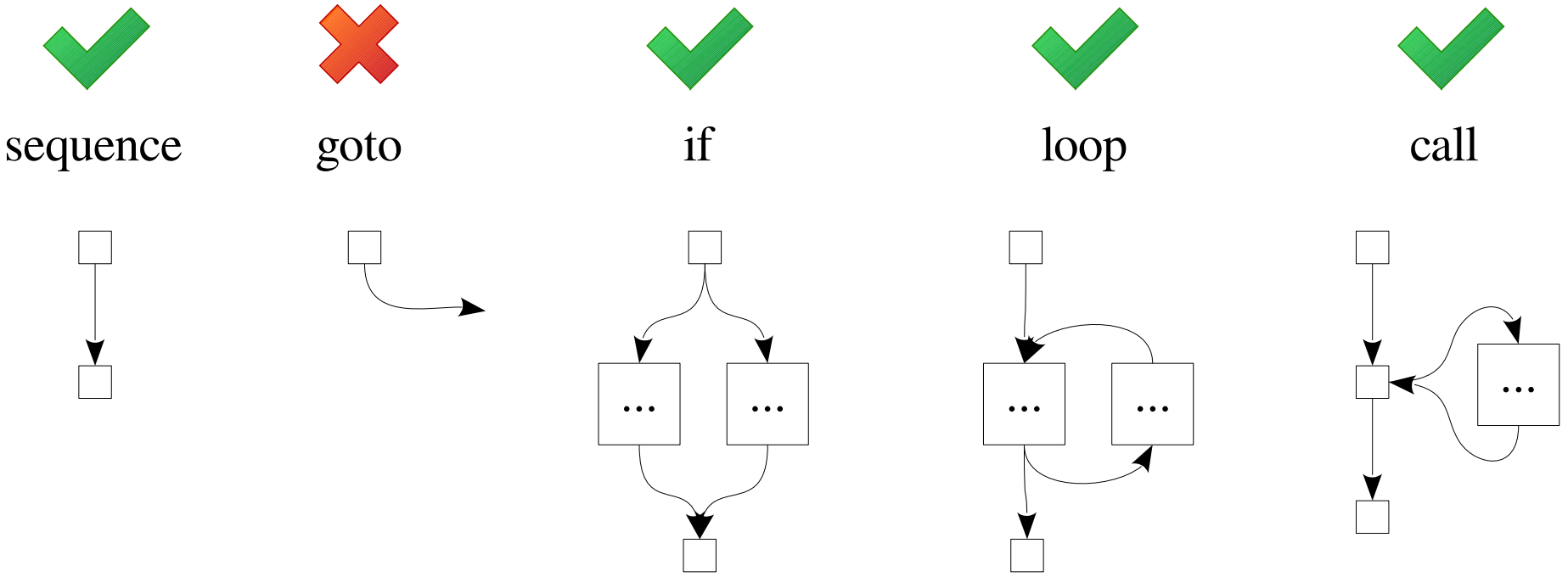
Are we there yet?

- OIO rewrite
- Continuations
- Strand → Fiber → Lightweight thread → Virtual thread
- Structured concurrency
- Scoped variables



Structured concurrency

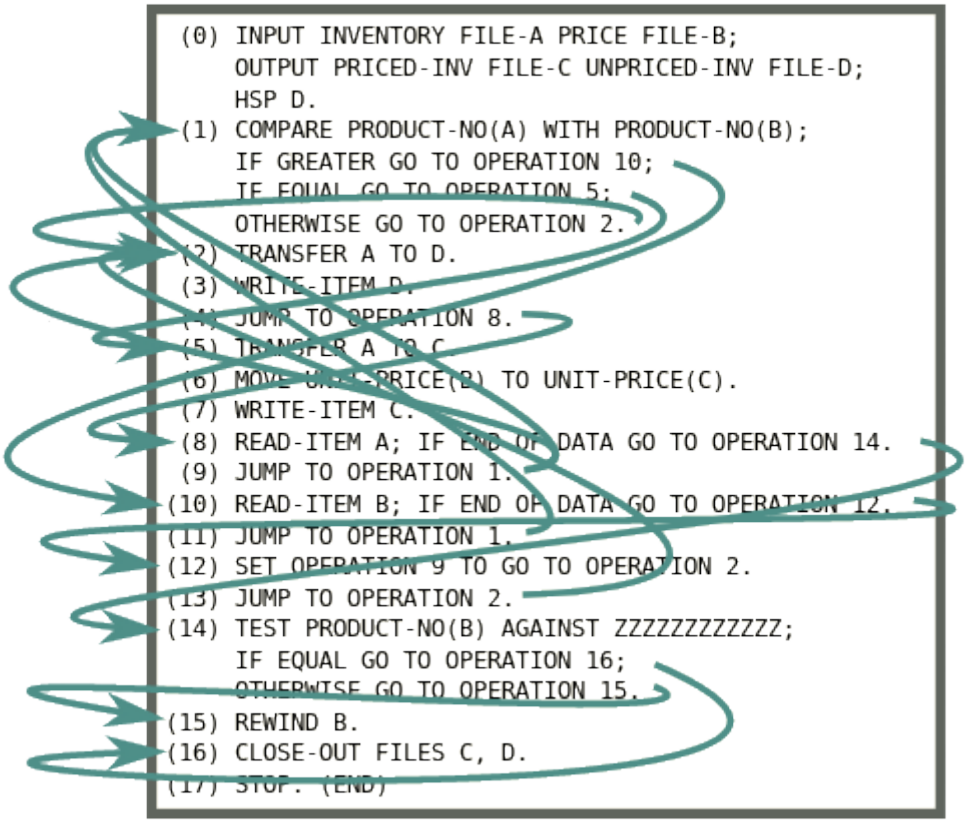
The curse of control flow constructs



Quoting from Martin Sustrik's "Structured Concurrency" talk in FOSDEM'19.

Is GOTO harmful?

(So are threads!)

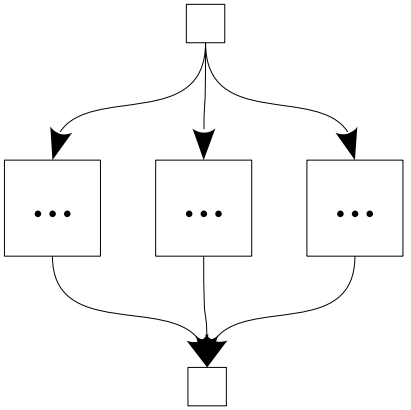


Quoting from Nathaniel J. Smith's "Notes on structured concurrency" blog post.

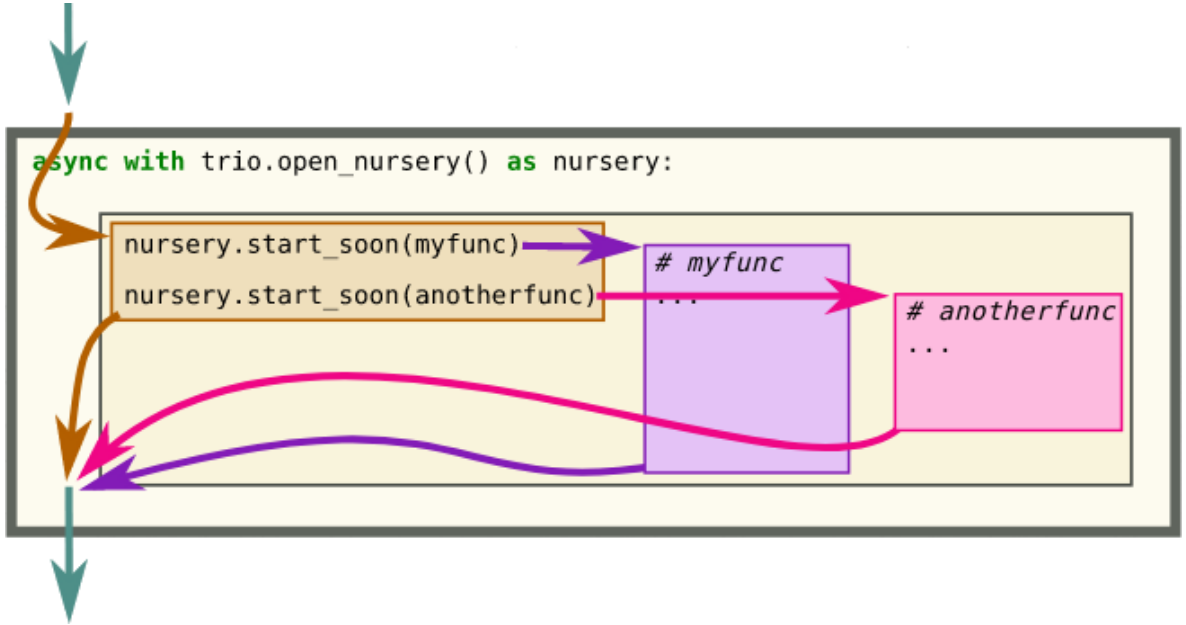
What does structured concurrency look like?

```
async with trio.open_nursery() as nursery:
    nursery.start_soon(myfunc)
    nursery.start_soon(anotherfunc)
<rest of program>
```

```
with open("my-file") as file_handle:
    ...
```




```
async with trio.open_nursery() as nursery:
    nursery.start_soon(myfunc)
    nursery.start_soon(anotherfunc)
    # myfunc
    ...
    # anotherfunc
    ...
<rest of program>
```

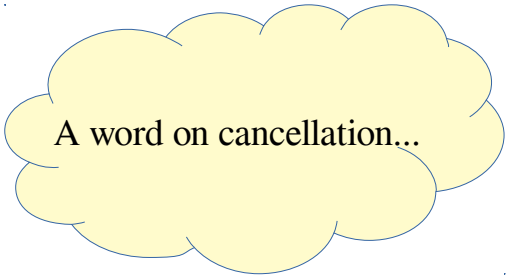


Quoting from Nathaniel J. Smith's "Notes on structured concurrency" blog post.

A glimpse of Trio

```
import trio

async def child1():
    print("  child1: started! sleeping now...")
    await trio.sleep(1)
    print("  child1: exiting!")

async def child2():
    print("  child2: started! sleeping now...")
    await trio.sleep(1)
    print("  child2: exiting!")

async def parent():
    print("parent: started!")
    async with trio.open_nursery() as nursery:
        print("parent: spawning child1...")
        nursery.start_soon(child1)

        print("parent: spawning child2...")
        nursery.start_soon(child2)

        print("parent: waiting for children to finish...")
        # -- we exit the nursery block here --
    print("parent: all done!")

trio.run(parent)
```

```
parent: started!
parent: spawning child1...
parent: spawning child2...
parent: waiting for children to finish...
  child2: started! sleeping now...
  child1: started! sleeping now...
    [... 1 second passes ...]
  child1: exiting!
  child2: exiting!
parent: all done!
```



Scoped variables

Don't we already have scoped variables?

```
public class ScopeDemo {

    private static final Logger LOGGER = LoggerFactory.getLogger(ScopeDemo.class);

    private final List<Runnable> tasks = new ArrayList<>();

    private final ThreadLocal<StringBuilder> stringBuilderRef =
            ThreadLocal.withInitial(StringBuilder::new);

    public void addTask(Runnable task) {
        tasks.add(task);
    }

    public static void main(String[] args) throws IOException {
        try (InputStream inputStream = new FileInputStream("/etc/passwd")) {
            int firstByte = inputStream.read();
            {
                int randomByte = (int) Math.abs(Math.random() * 0xFF);
                firstByte += randomByte;
            }
        }
    }

}
```

What is the scope of a ThreadLocal?

Static-vs-Dynamic scoping

```
> x=1

> function g() {
    echo $x;
    x=2;
  }

> function f() {
    local x=3;
    g;
  }

> f

> echo $x
```

What do these two statements output?

Dynamic scoping and structured concurrency

```
> color=null

> function terrier(nursery) {
    echo “Terrier sees $color.”;
  }

> function dog(nursery) {
    echo “Dog sees $color.”
    color=”black-and-white”;
    nursery.schedule(terrier)
  }

> function cat(nursery) {
    echo “Cat sees $color.”
  }

> with nursery {
    color=”colorful”
    nursery.schedule(dog)
    nursery.schedule(cat)
  }
```

What does this statement output?

Conclusions

- Loom will radically change I/O composition in JVM.
- Structured concurrency and scoped variables will supplement that ease.

Thank you!

(Questions?)

Volkan Yazıcı
https://vlkan.com
@yazicivo