Kotlin Coroutines and the Order of the Concurrency

by Artur Radiuk

Example problem

Submit order

1. Fetch offers for cart
2. Fetch user’s subscriptions
3. Fetch delivery methods
4. Fetch payment methods
5. Validate purchase
6. Place order

Example problem

What options do we have?

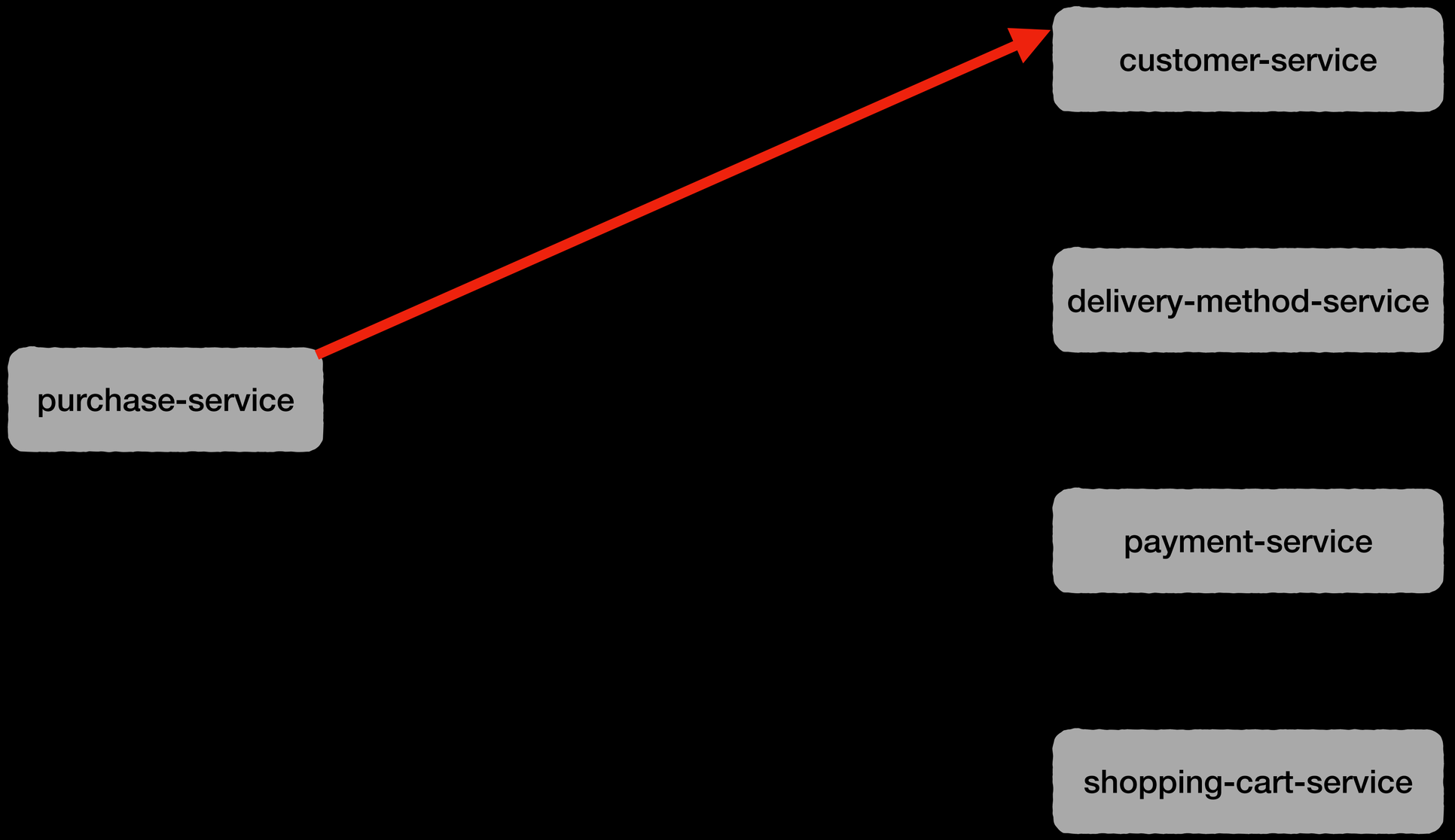
Synchronous solution

```
fun submitOrder(order: Order) {
    val userSubscriptions = fetchUserSubscriptions(userId = order.userId)
    val availableDeliveryMethods = fetchDeliveryMethods(cartId = order.cartId, userSubscriptions)

    val cartOffers = fetchCartOffers(cartId = order.cartId)
    val availablePaymentMethods = fetchPaymentMethods(cartId = order.cartId)

    val orderData = OrderData(
        cartOffers = cartOffers,
        availableDeliveryMethods = availableDeliveryMethods,
        availablePaymentMethods = availablePaymentMethods,
        userSubscriptions = userSubscriptions
    )

    validateOrderData(order = order, orderData = orderData)
    placeOrder(order)
}
```

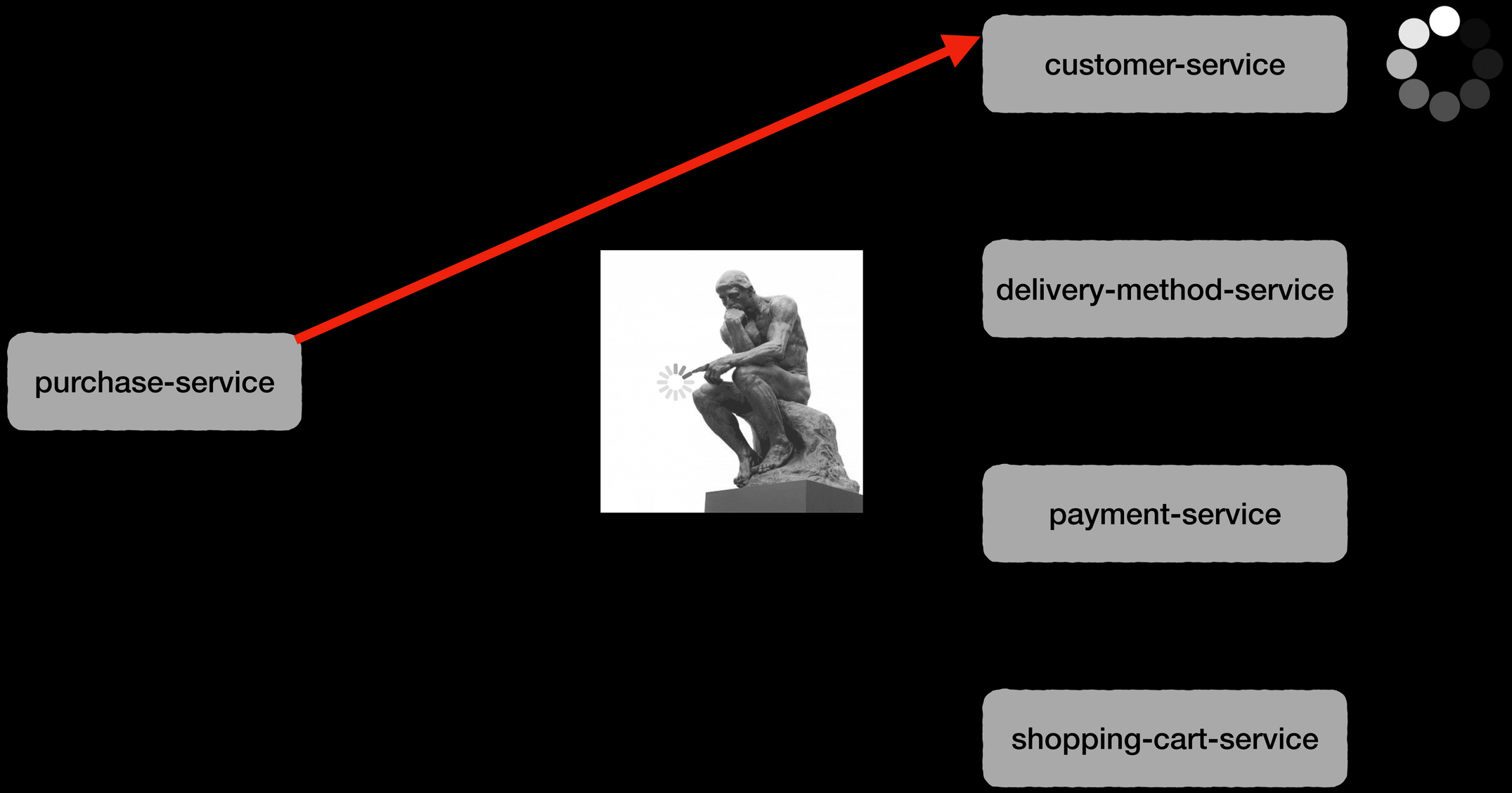
Synchronous solution

But how does it work?

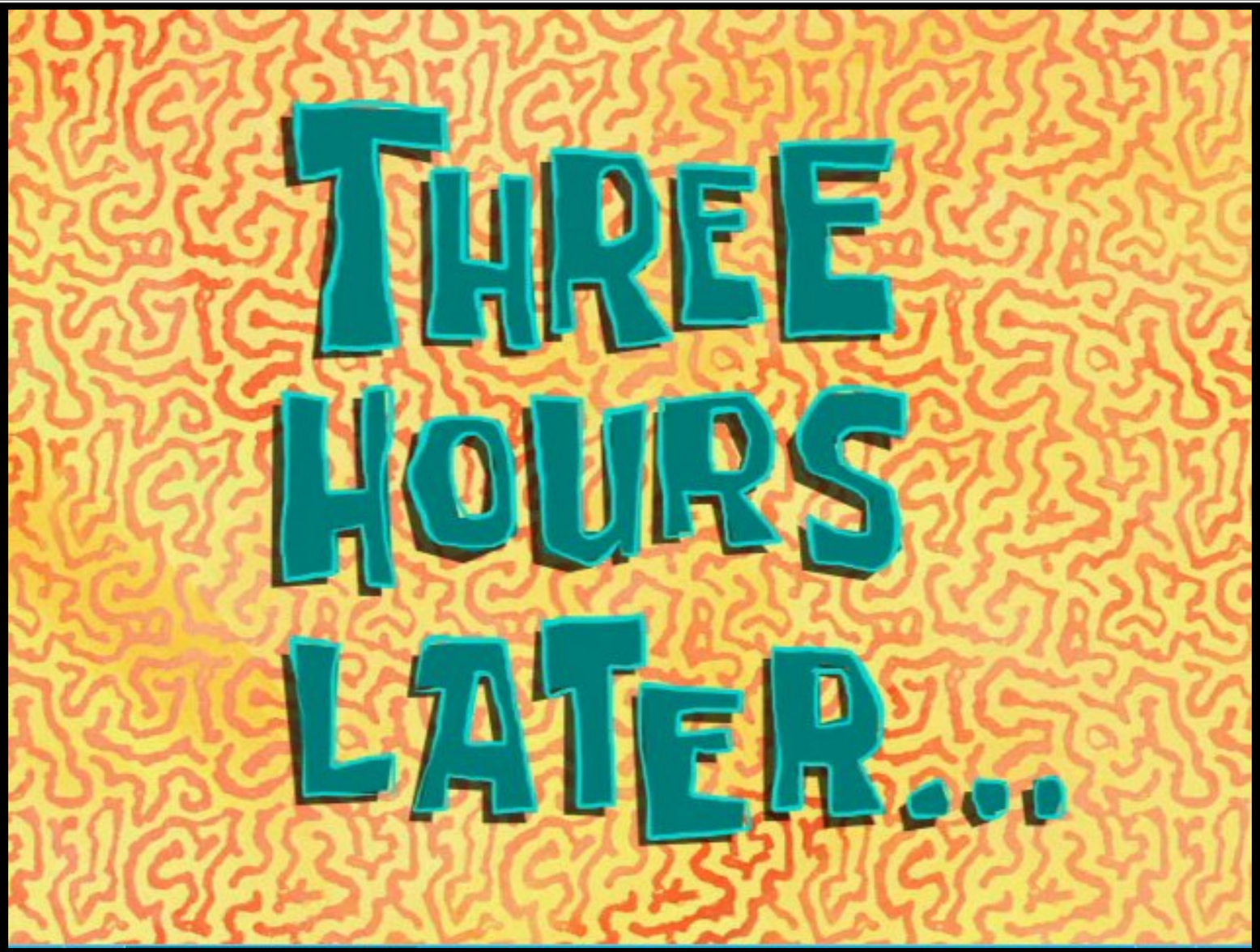
Synchronous solution



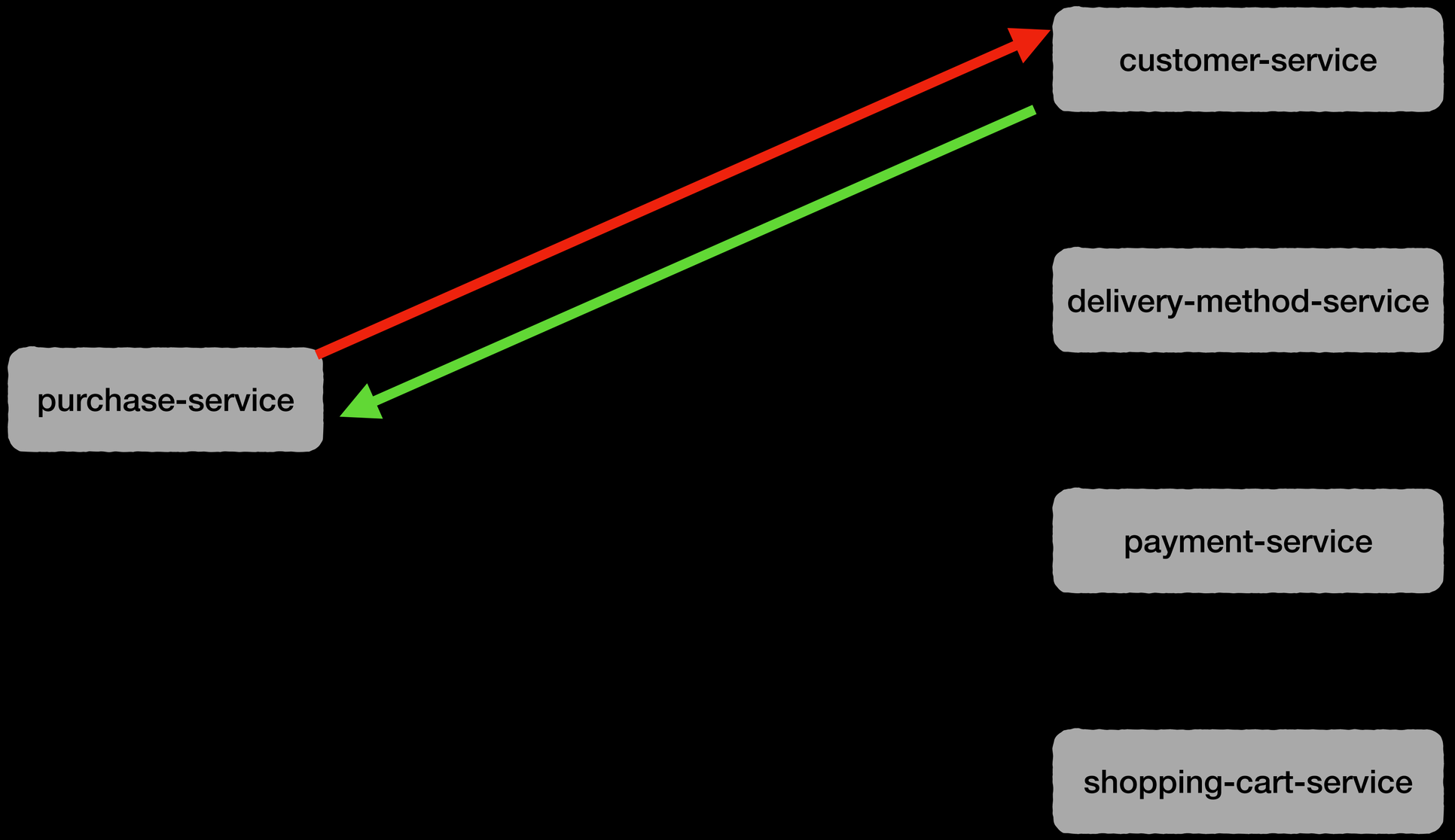
Synchronous solution



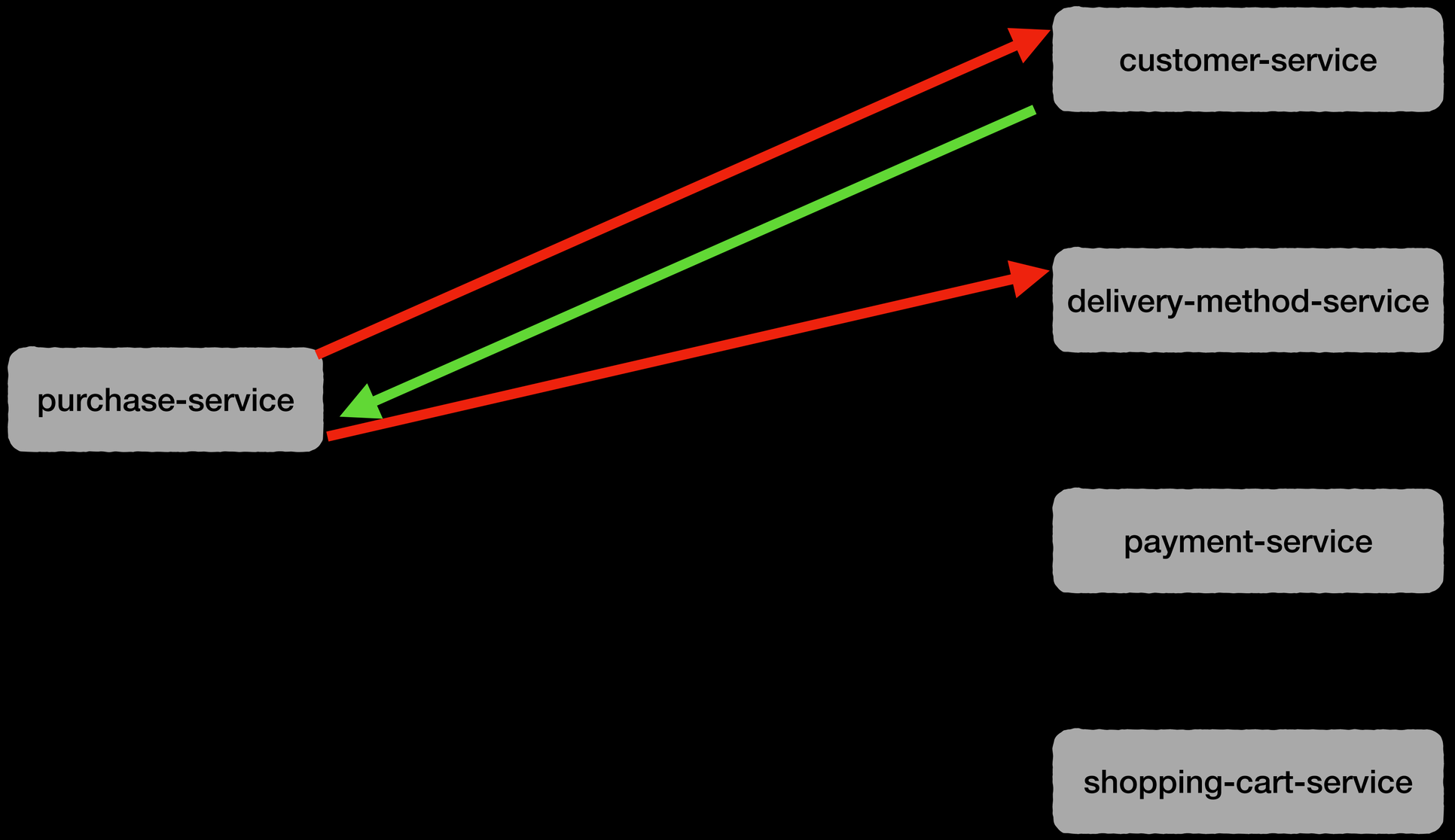
Synchronous solution



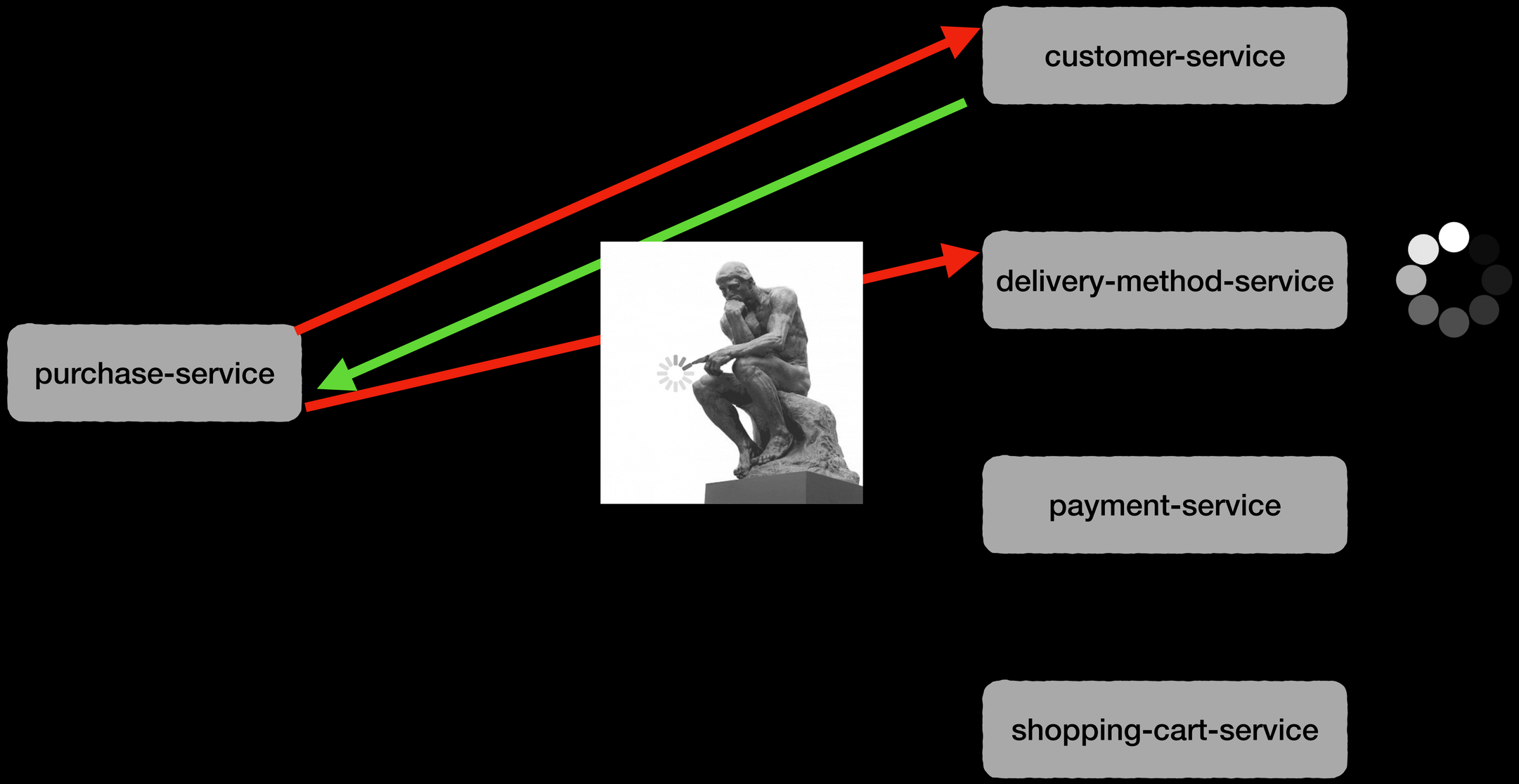
Synchronous solution



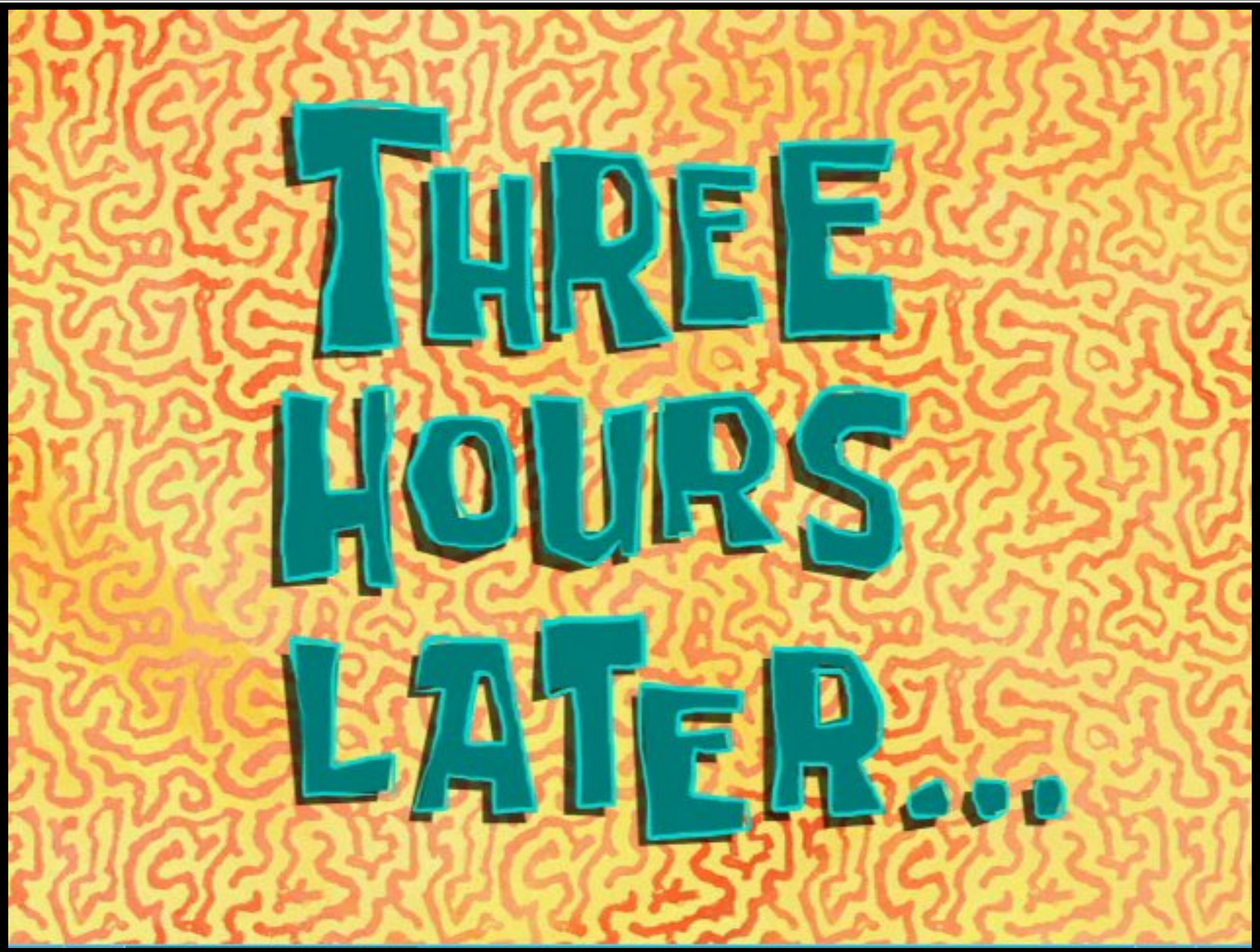
Synchronous solution



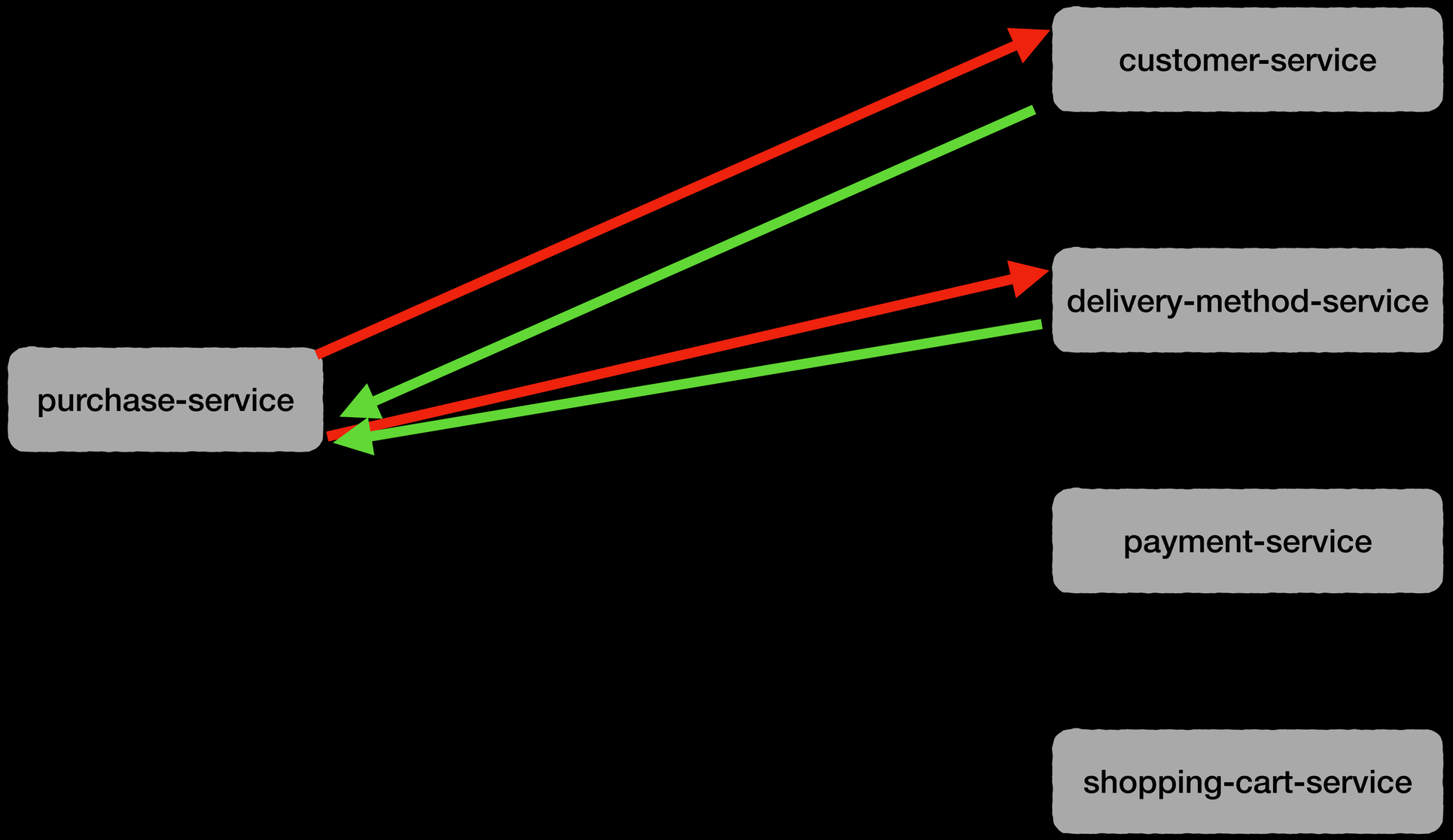
Synchronous solution



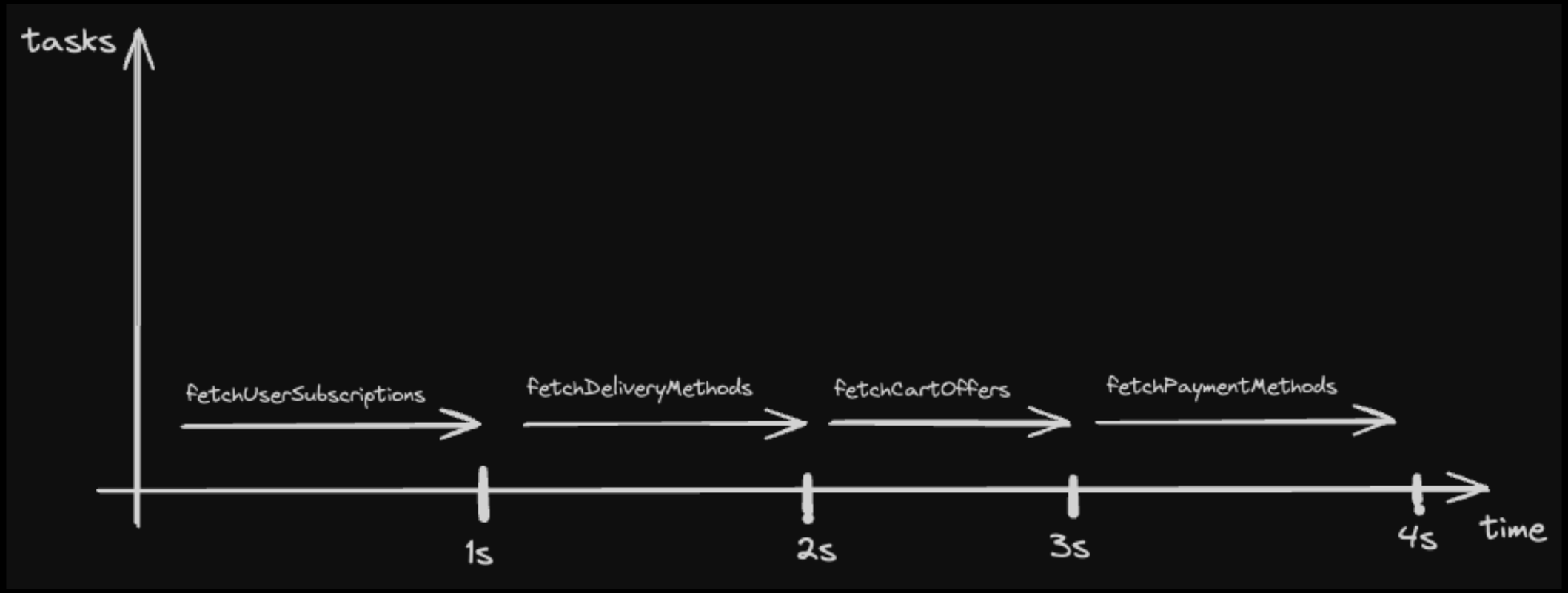
Synchronous solution



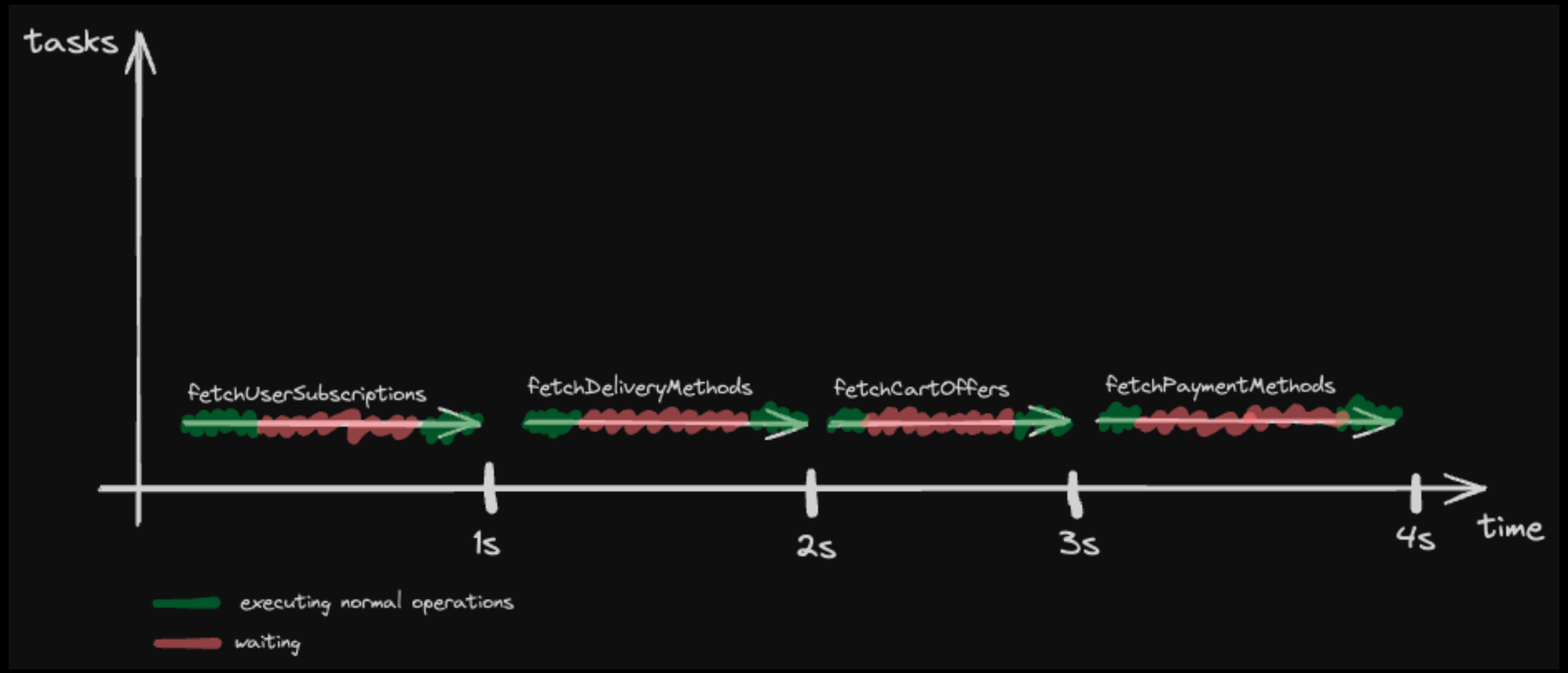
Synchronous solution



Synchronous solution



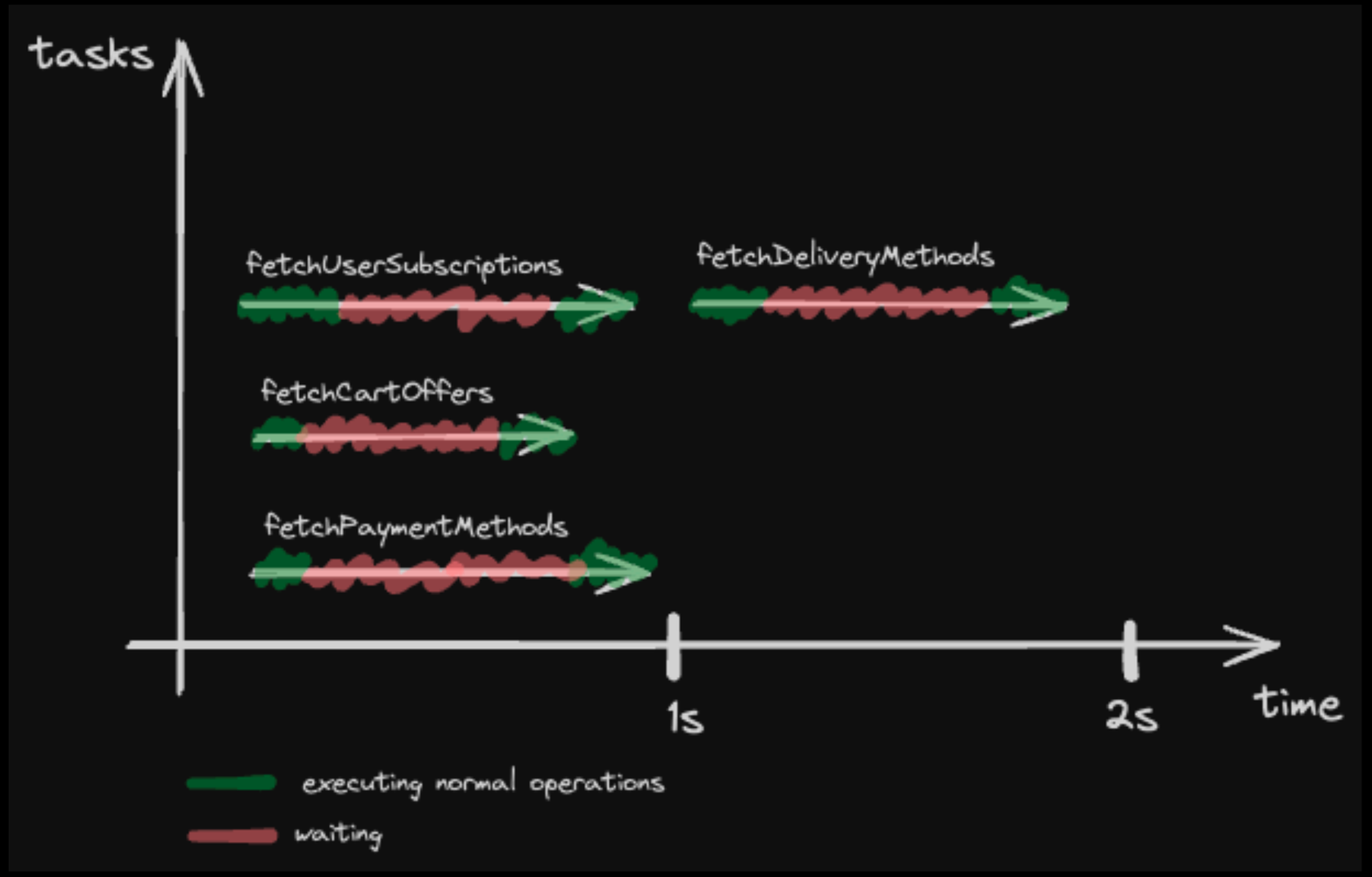
Synchronous solution



Synchronous solution

Maybe we can make it better?

Desired non-blocking solution



Multithreading solution

What about multithreading?

threads, executors etc.

Multithreading solution

Callback solution

```
fun submitOrder(order: Order) {
    val userSubscriptionsCallback = fetchUserSubscriptions(userId = order.userId)
    val subscriptionsToDeliveryMethods = userSubscriptionsCallback.future
        .thenApply {
            it to fetchDeliveryMethods(cartId = order.cartId, it).future
        }

    val cartOffers = fetchCartOffers(cartId = order.cartId)
    val availablePaymentMethods = fetchPaymentMethods(cartId = order.cartId)

    val (userSubscriptions, deliveryMethodsFuture) = subscriptionsToDeliveryMethods.join()
    val orderData = OrderData(
        cartOffers = cartOffers.future.join(),
        availableDeliveryMethods = deliveryMethodsFuture.join(),
        availablePaymentMethods = availablePaymentMethods.future.join(),
        userSubscriptions = userSubscriptions
    )

    Common.validateOrderData(order = order, orderData = orderData)

    client.dispatcher.executorService.shutdown()
    Common.placeOrder(order)
}
```

Callback solution

What about:

- error handling?
- cancelling?

Callback solution

Better, but:

- Accidental complexity
- Cancelling is hard
- Error handling is hard

Reactor solution

```
fun submitOrder(order: Order) {
    val userSubscriptionsMono = fetchUserSubscriptions(userId = order.userId)
    val userSubscriptionsTo = userSubscriptionsMono.map {
        it to fetchDeliveryMethods(cartId = order.cartId, subscriptions = it)
    }

    val cartOffers = fetchCartOffers(cartId = order.cartId)
    val availablePaymentMethods = fetchPaymentMethods(cartId = order.cartId)

    val (userSubscriptions, deliveryMethodsMono) = userSubscriptionsTo.block()!!
    val zip = Mono.zip(
        cartOffers, deliveryMethodsMono, availablePaymentMethods
    ).block()

    val orderData = OrderData(
        cartOffers = zip.t1,
        availableDeliveryMethods = zip.t2,
        availablePaymentMethods = zip.t3,
        userSubscriptions = userSubscriptions
    )

    Common.validateOrderData(order = order, orderData = orderData)

    Common.placeOrder(order)
}
```

Reactor solution

Event more better, but:

- accidental complexity
- external tool (versions, compatibility, inner implementation…)

Reactor solution

What about:

- error handling?
- cancelling?

Example problem

What if submit purchase must be asynchronous?

Coroutines solution

```
suspend fun submitOrder(order: Order) = coroutineScope {
    val userSubscriptions = async { fetchUserSubscriptions(userId = order.userId) }
    val availableDeliveryMethods = async { fetchDeliveryMethods(cartId = order.cartId, userSubscriptions.await()) }

    val cartOffers = async { fetchCartOffers(cartId = order.cartId) }
    val availablePaymentMethods = async { fetchPaymentMethods(cartId = order.cartId) }

    val orderData = OrderData(
        cartOffers = cartOffers.await(),
        availableDeliveryMethods = availableDeliveryMethods.await(),
        availablePaymentMethods = availablePaymentMethods.await(),
        userSubscriptions = userSubscriptions.await()
    )

    Common.validateOrderData(order = order, orderData = orderData)
    Common.placeOrder(order)
}
```

Coroutines solution

- language built-in, ubiquitous API
- flexible
- minimum accidental complexity
- asynchronous and concurrent APIs in one
- has default and configurable cancelation mechanism
- has default and configurable error handling mechanism

Coroutine definition

“A coroutine is an instance of a suspendable computation. It is conceptually similar to a thread, in the sense that it takes a block of code to run that works concurrently with the rest of the code. However, a coroutine is not bound to any particular thread. It may suspend its execution in one thread and resume in another one.”

Suspending function

What is suspending function?

- Function that can be ***suspended*** and ***resumed*** at ***some*** point in future

Requirements:

- suspend function must be called by coroutine or by another suspending function
- ‘*suspend*’ keyword

Suspending function

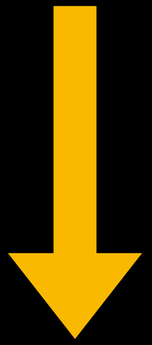
Show code snippets from package ‘**n1_suspendingfunction**’

Suspension under the hood

```
suspend fun sampleSuspendableFunction() {
    println("Before delay")
    delay(1000)
    println("After delay")
}
```

Suspension under the hood

```
1 suspend fun sampleSuspendableFunction()
```



```
1 suspend fun sampleSuspendableFunction(continuation: Continuation<*>): Any
```

Suspension under the hood

Continuation

- is passed implicitly as a parameter from suspending function to suspending function
- every suspending function has its own implementation of continuation that stores ALL parents continuations
- allows to resume execution of suspended function
- stores coroutine context (config)

Suspension under the hood

Show code snippets from package ‘**n2_suth**’

Suspension under the hood

What if function has state?

Suspension under the hood

```
suspend fun main() {
    firstSuspendableFunction()
}

suspend fun firstSuspendableFunction(): Int {
    var someNumber = 4
    println("First before, someNumber: ${someNumber}")
    secondSuspendableFunction()
    someNumber = 5
    println("First after, someNumber: ${someNumber}")
    return someNumber
}

suspend fun secondSuspendableFunction(): String {
    var someString = "hello"
    println("Second before, someString: ${someString}")
    delay(1000)
    someString = "hello, world"
    println("Second after, someString: ${someString}")
    return someString
}
```

Suspension under the hood

```
SecondSuspendableFunction_Continuation {
    label: 0,
    someString: "hello",

    parentContinuation: FirstSuspendableFunction_Continuation {
            label: 1,
            someNumber: 4
    }
}
```

Suspension under the hood

- Suspending functions is kind of state machine with possible states before and after every suspension point
- Continuation of current suspending function wraps parent’s continuation

Coroutine builders

Suspending function should be called only by another suspending function or by coroutine.

So, how to start a coroutine?

To start coroutine we need to use **coroutine builder**.

- runBlocking, runTest
- launch
- async

Coroutine builders - launch/async

launch/async

- extension function on the CoroutineScope interface (part of the mechanism called “structured concurrency”
- similar to starting thread in daemon mode (threads_vs_launch.kt)
- suspendCoroutine doesn’t block the thread, just suspends the coroutine

Coroutine builders - runBlocking

runBlocking

- top level coroutine builder
- provides coroutine scope for its children
- blocks execution of the thread on which it was called

Coroutine builders

Every coroutine:

- is started inside specific coroutine scope
- has it’s own inherited coroutine context

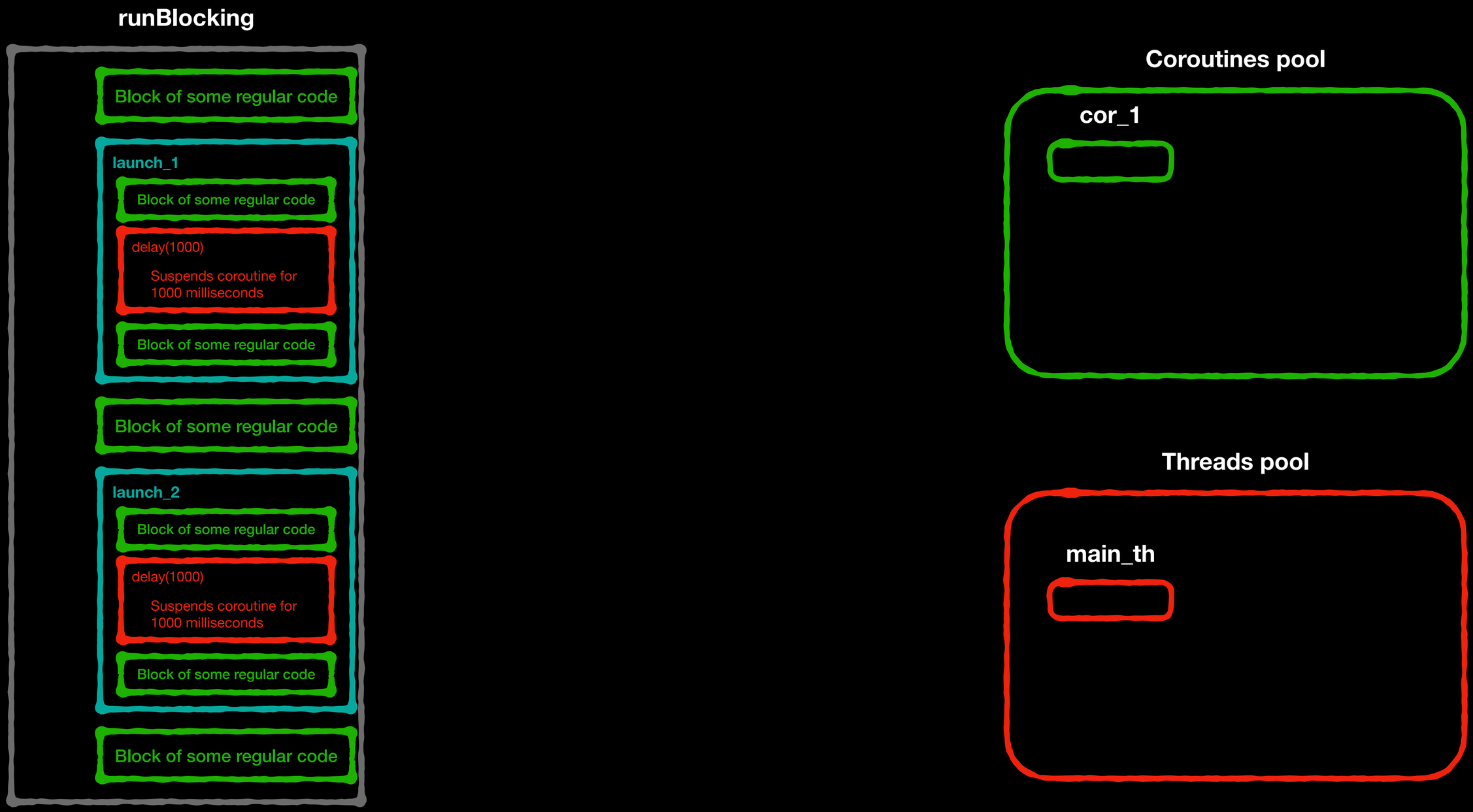
Structured concurrency

`structured concurrency` - mechanism of building a relationships between the parent coroutine and a child coroutine. CoroutineScope defines the lifetime of the coroutine and its children.

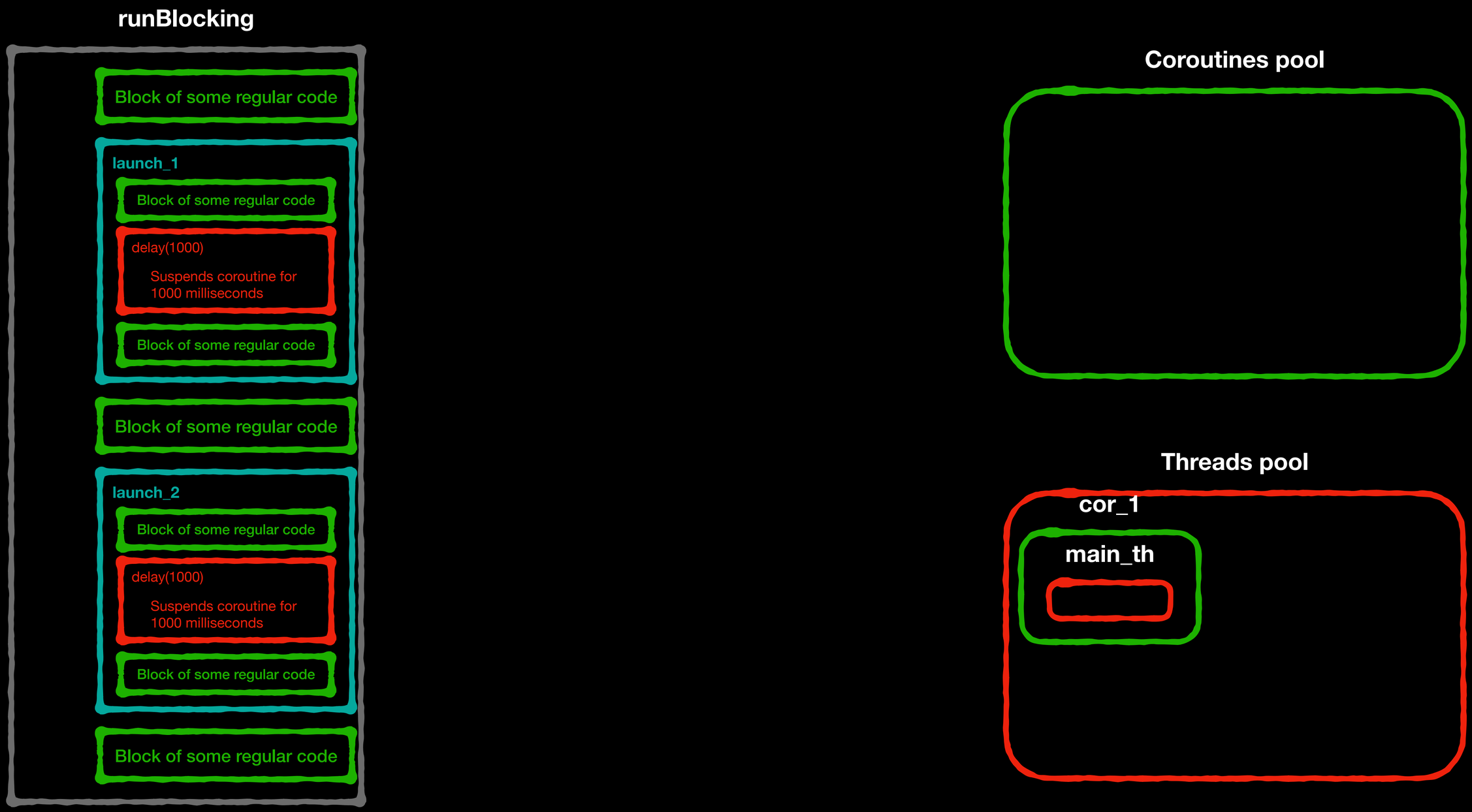
Parent coroutine:

- provides scope for its children;
- parent suspends (not blocks!) until all the children are completed;
- children inherit context (configurations) from their parents;
- when the parent is cancelled, its children coroutines are cancelled as well;
- when a child is destroyed it also destroys its parent;

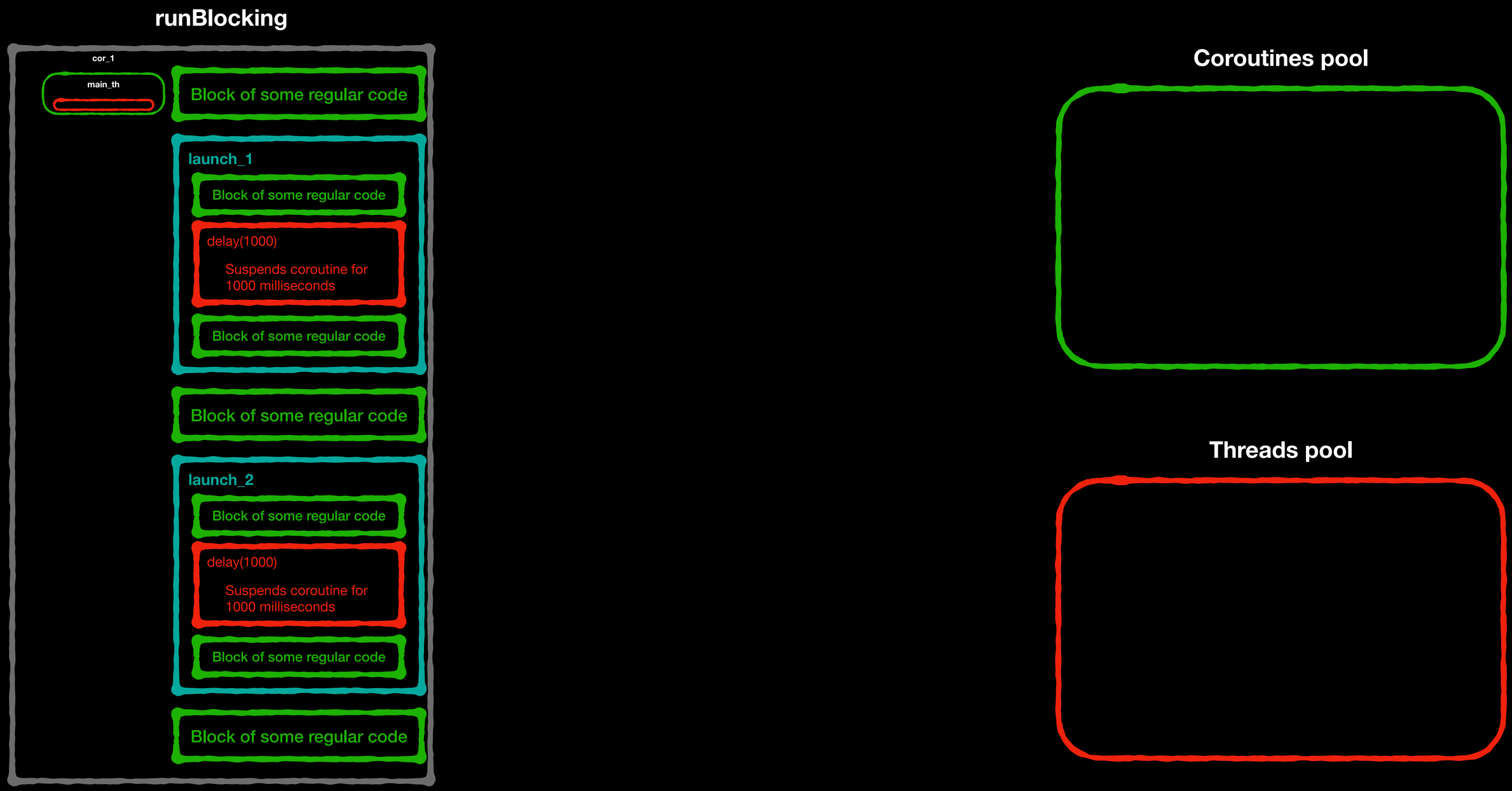
Structured concurrency - animation



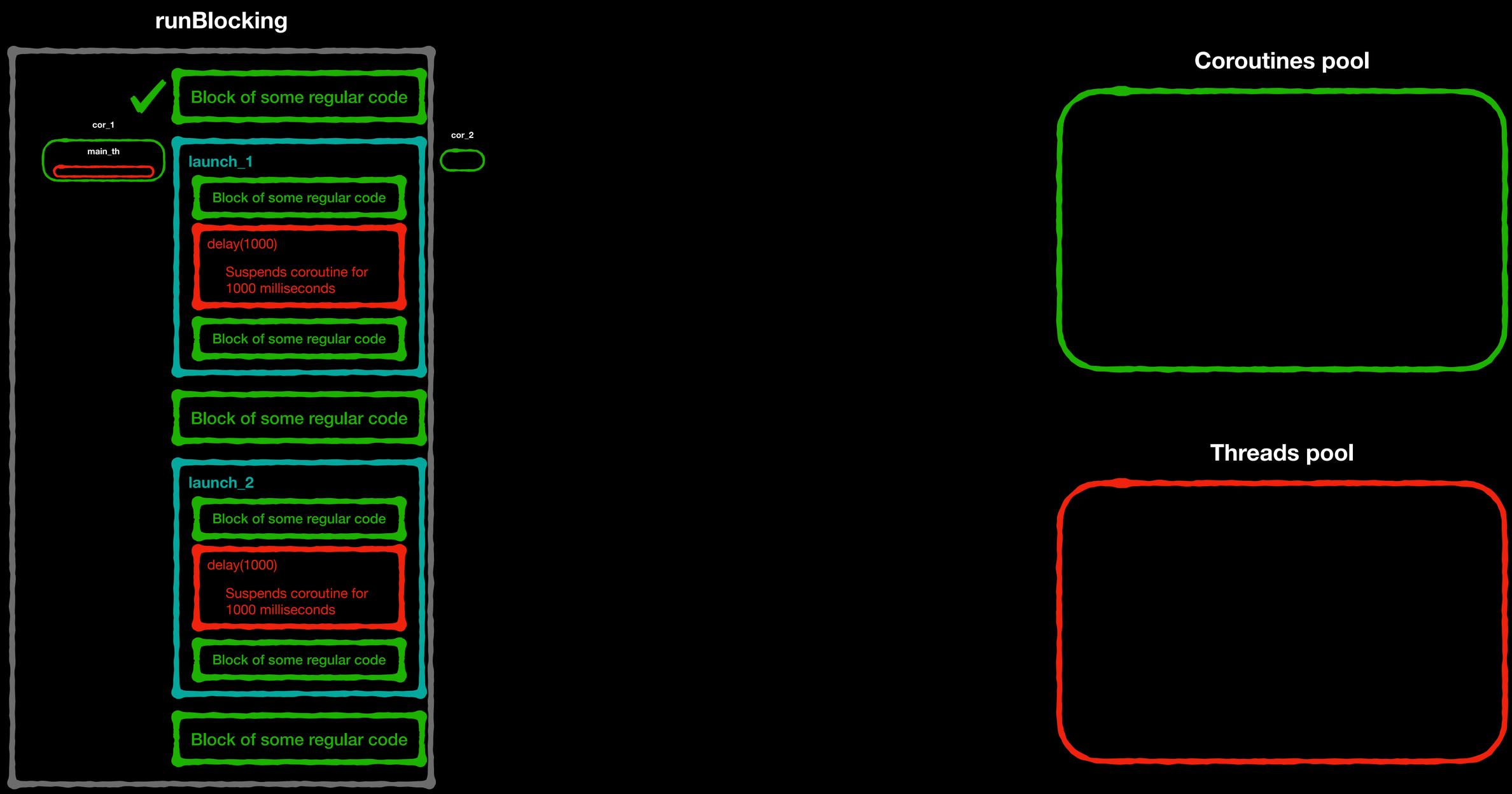
Structured concurrency - animation



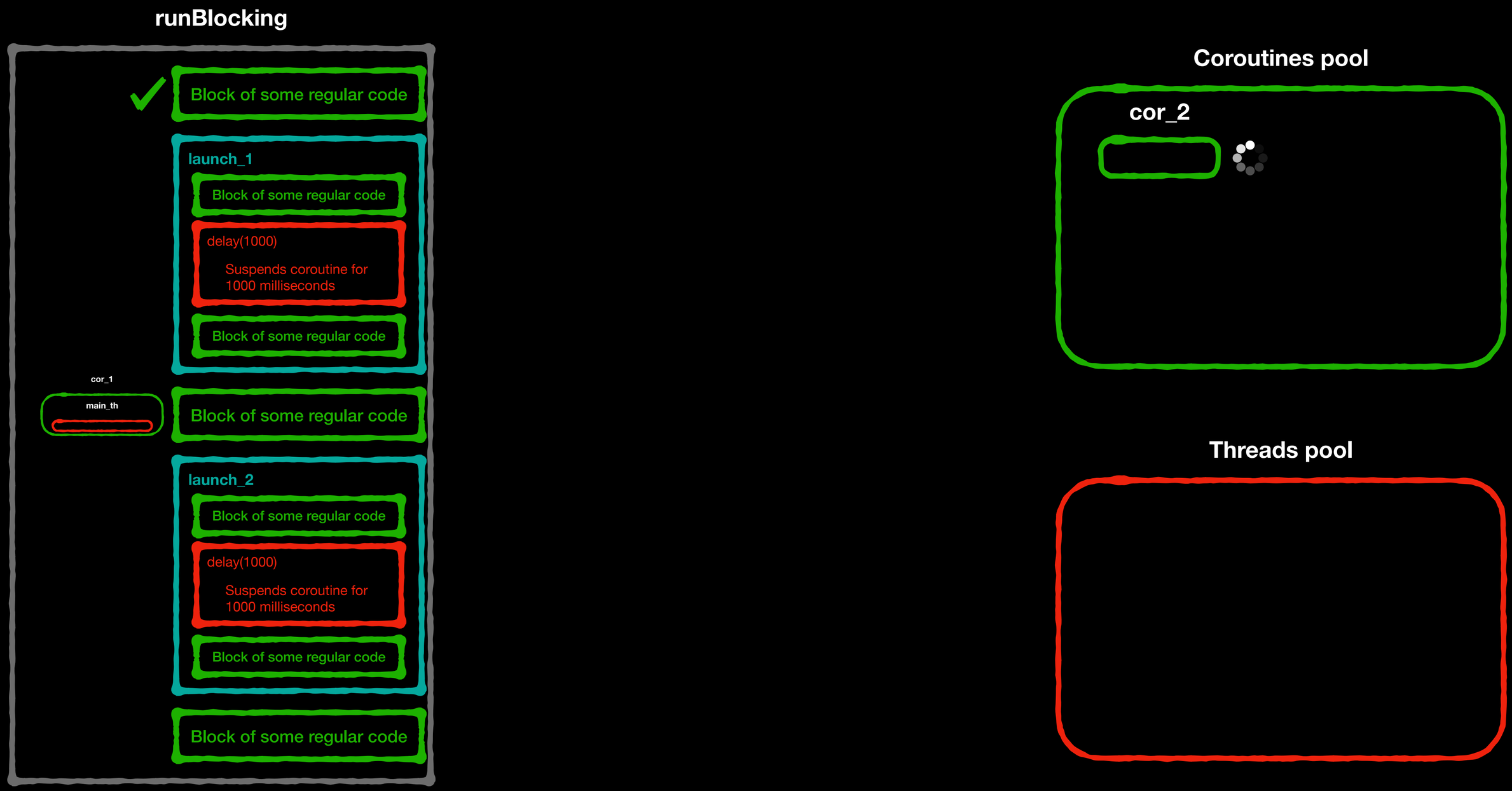
Structured concurrency - animation



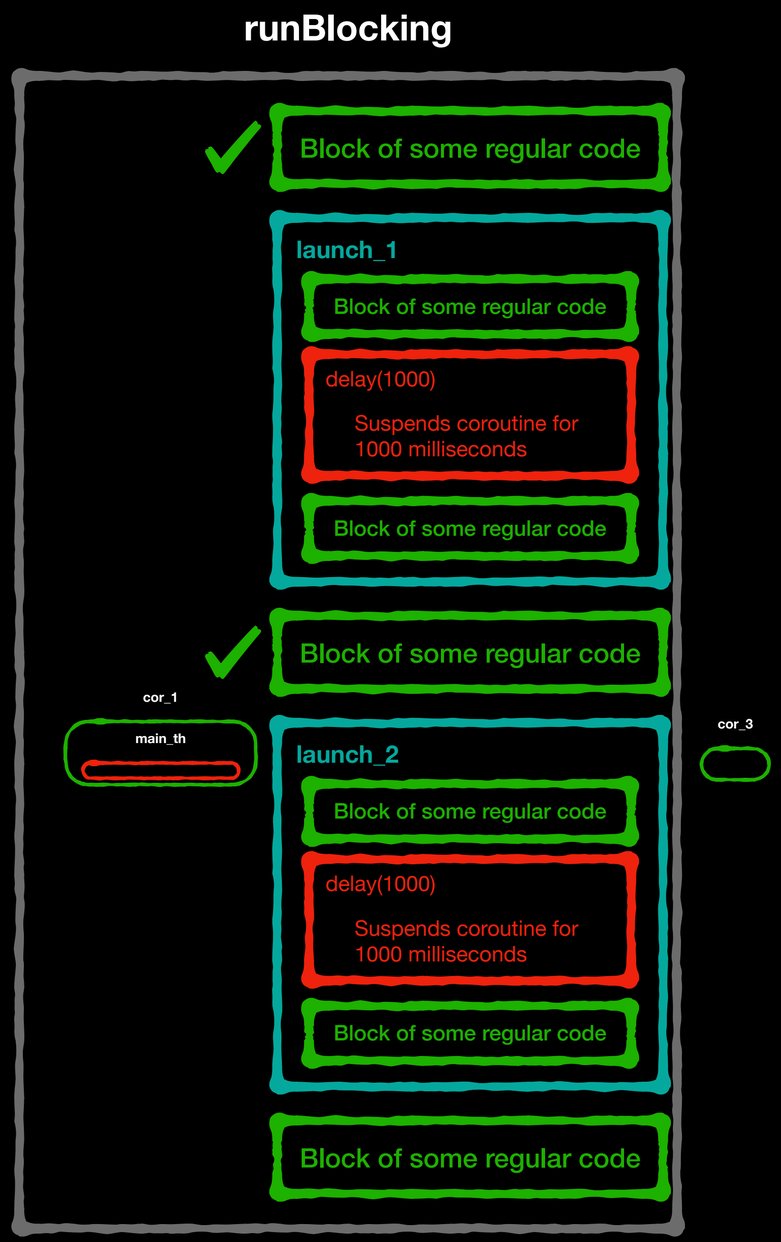
Structured concurrency - animation

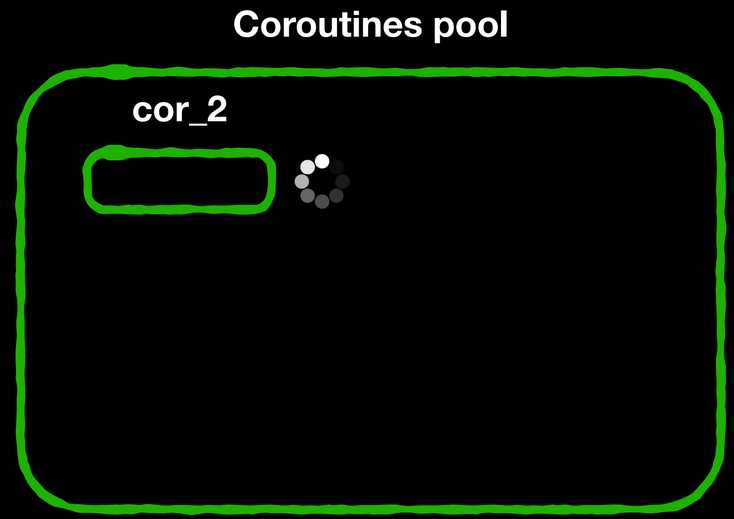


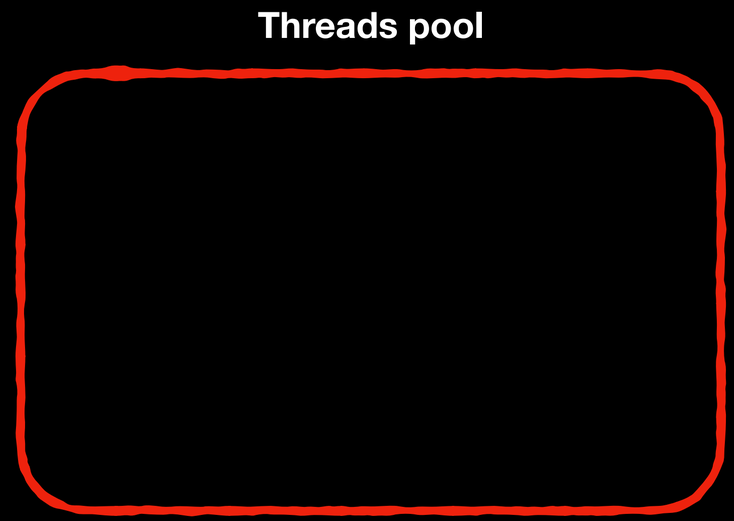
Structured concurrency - animation



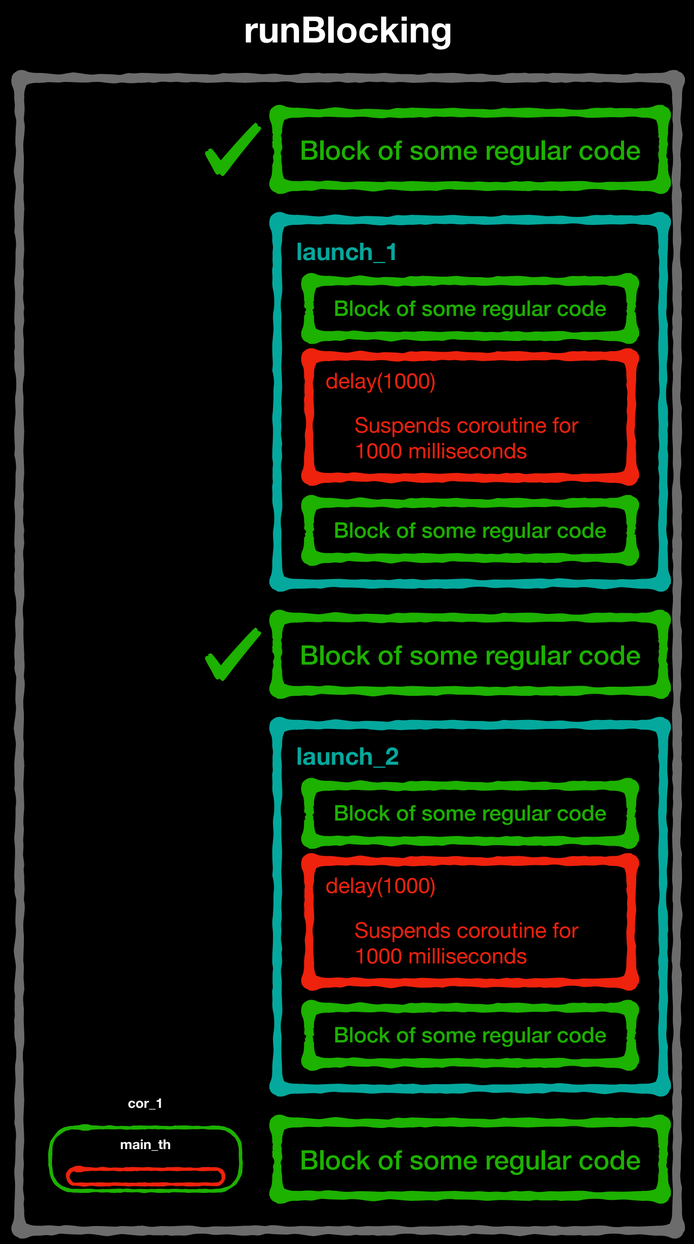
Structured concurrency - animation

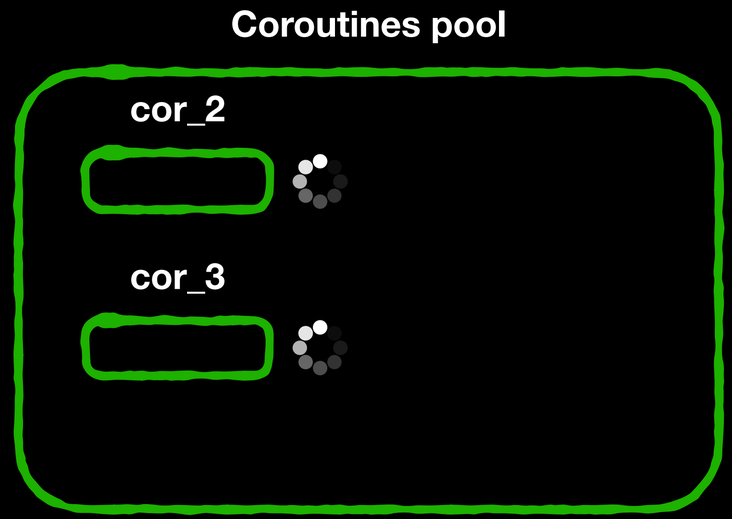


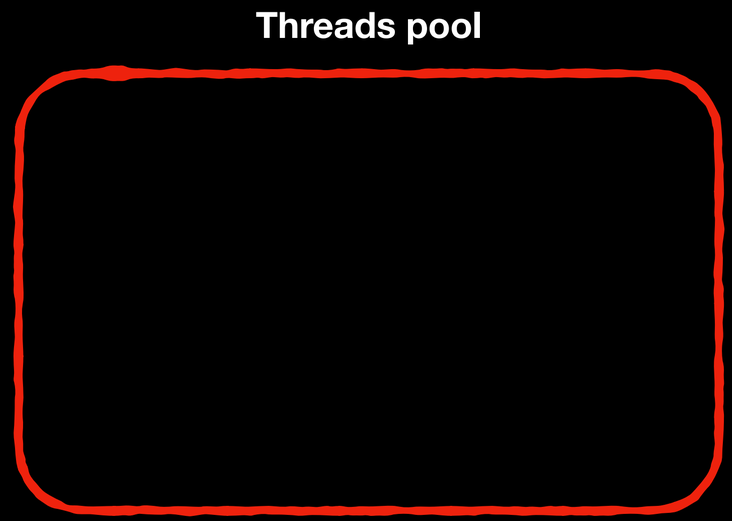




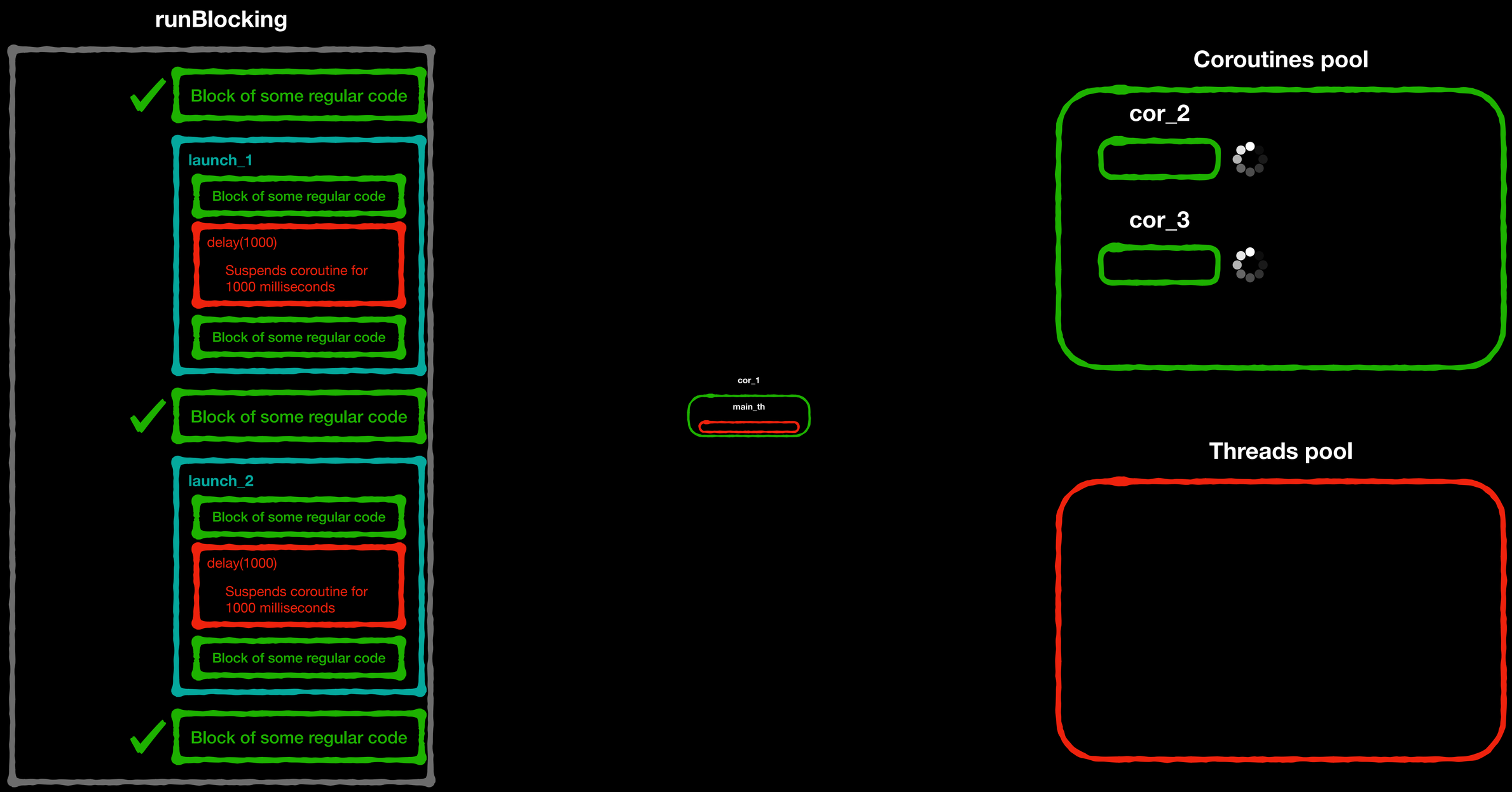
Structured concurrency - animation



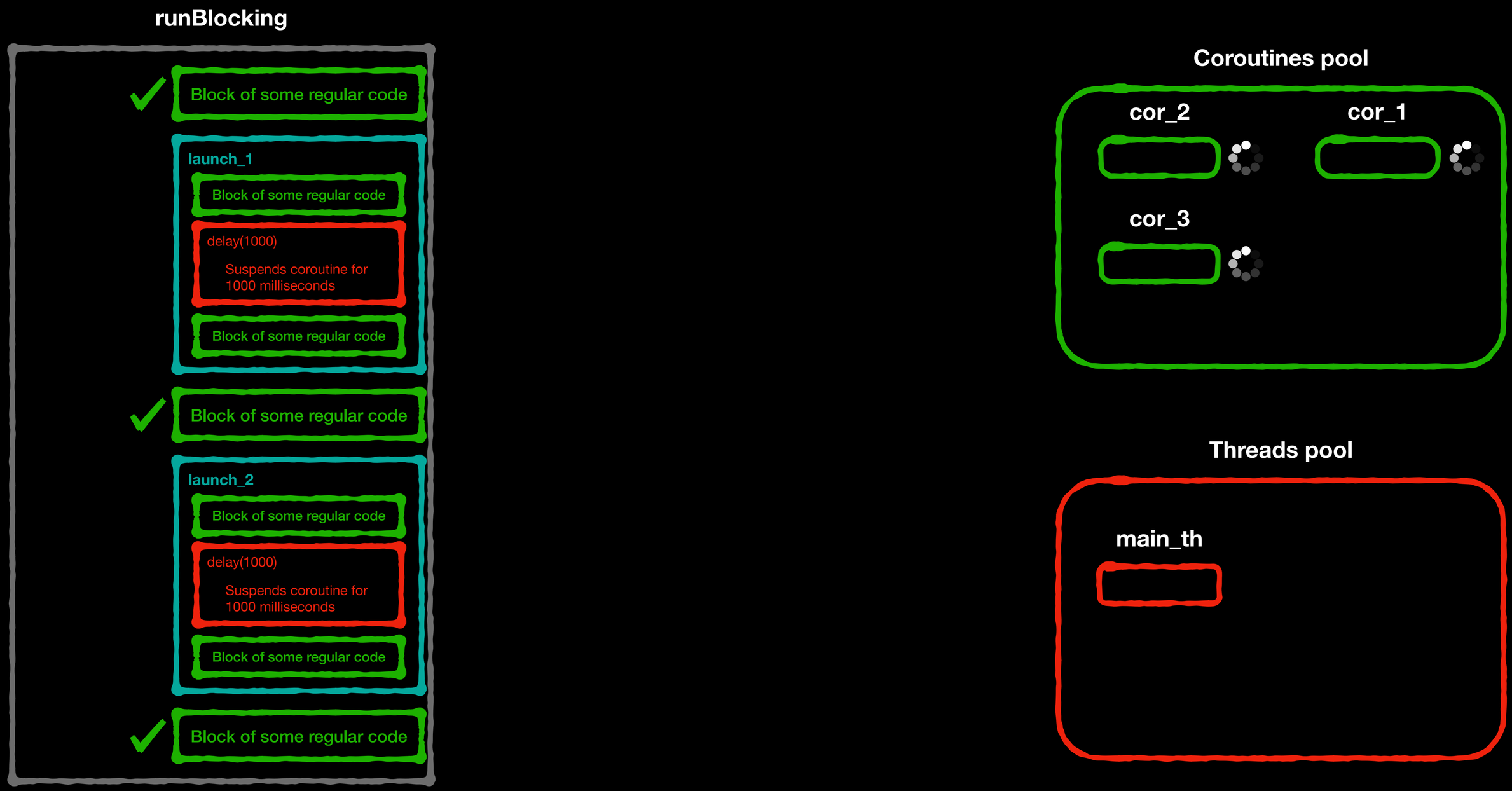




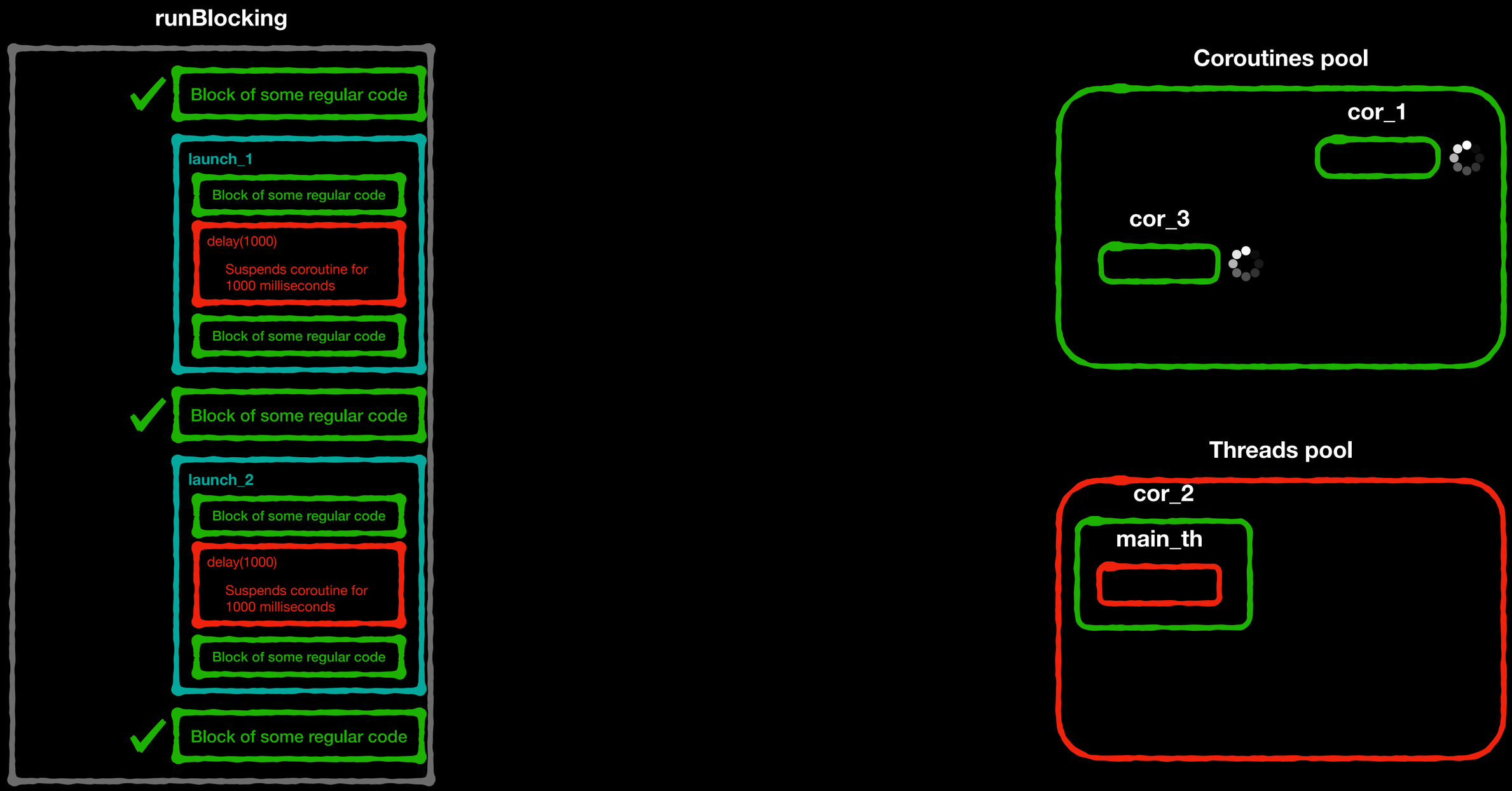
Structured concurrency - animation



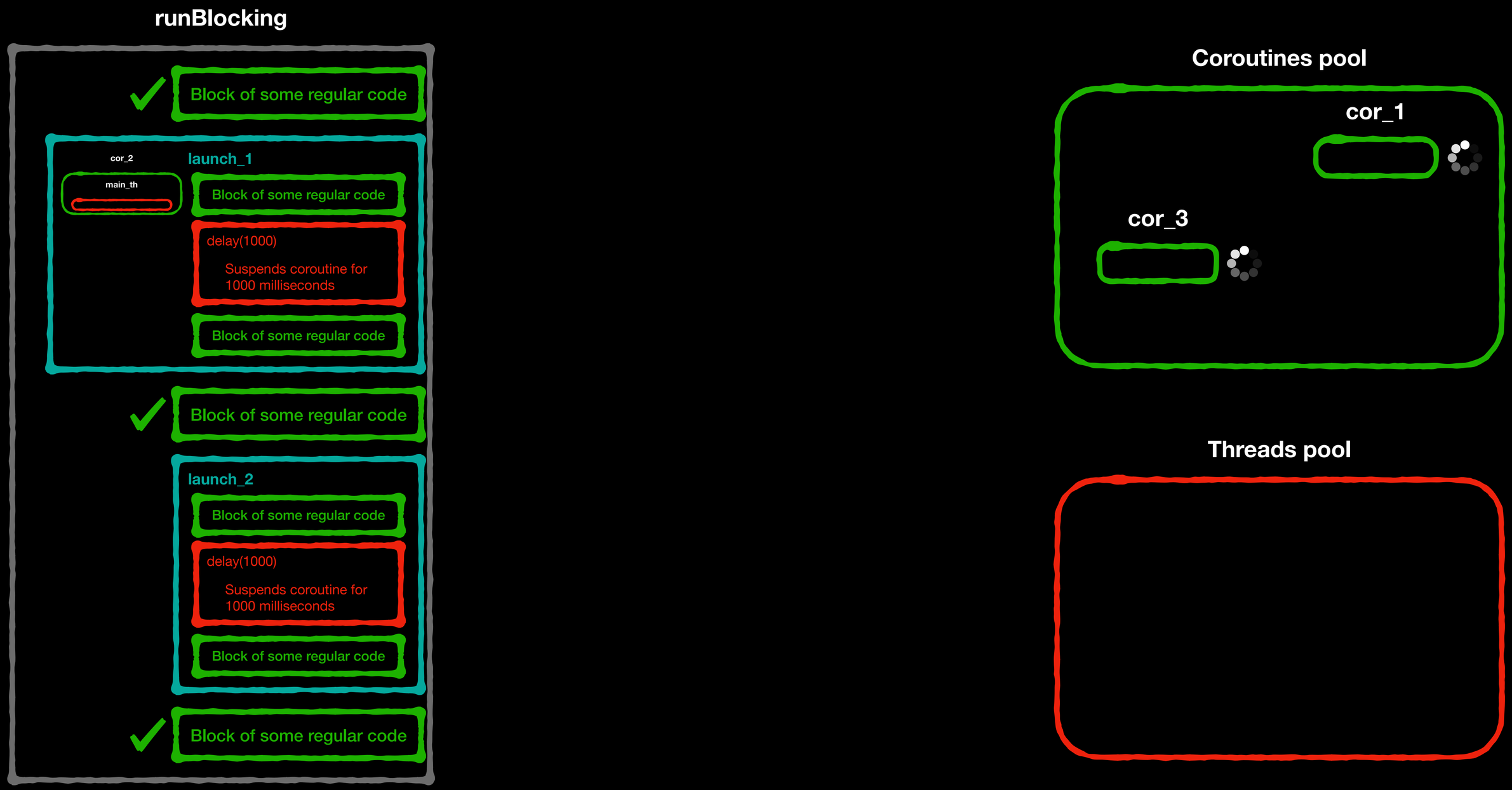
Structured concurrency - animation



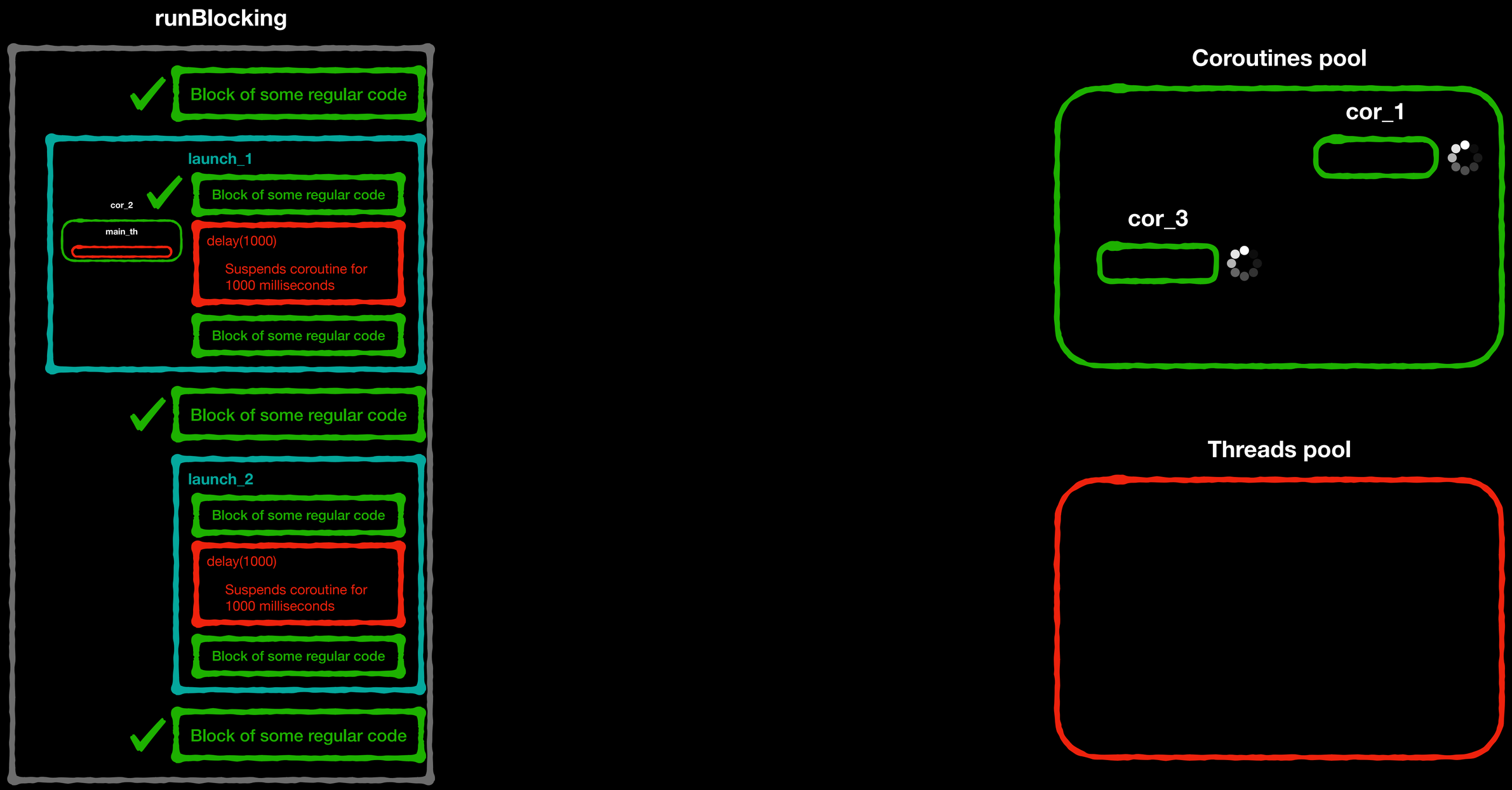
Structured concurrency - animation



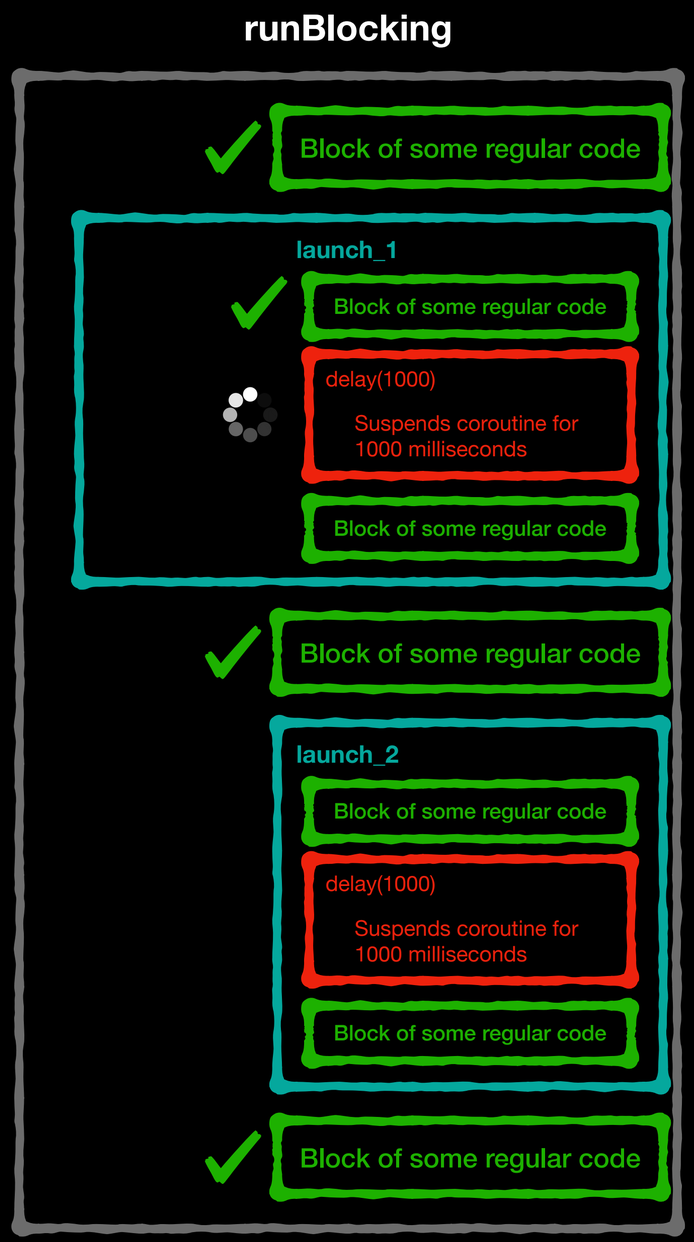
Structured concurrency - animation

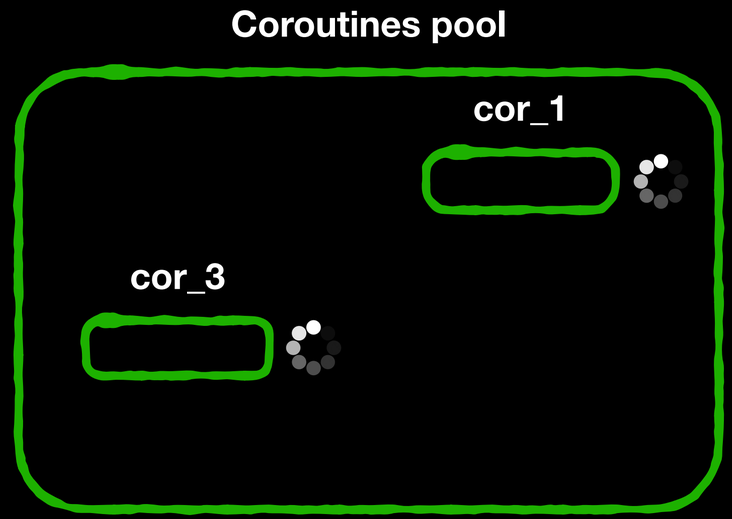


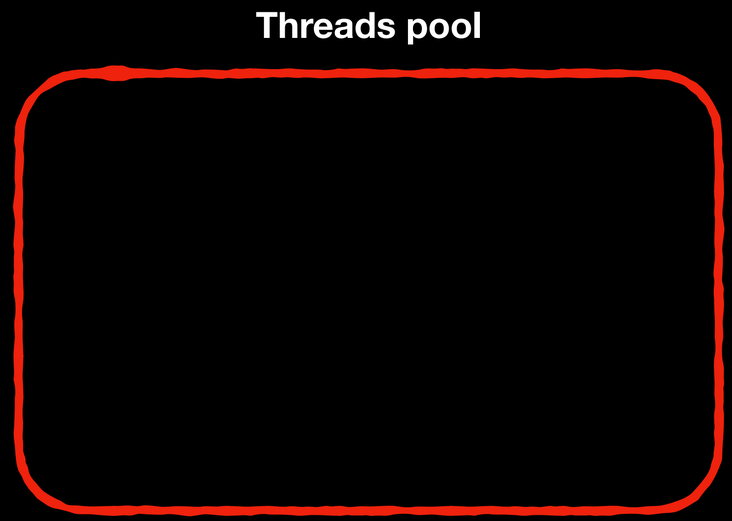
Structured concurrency - animation



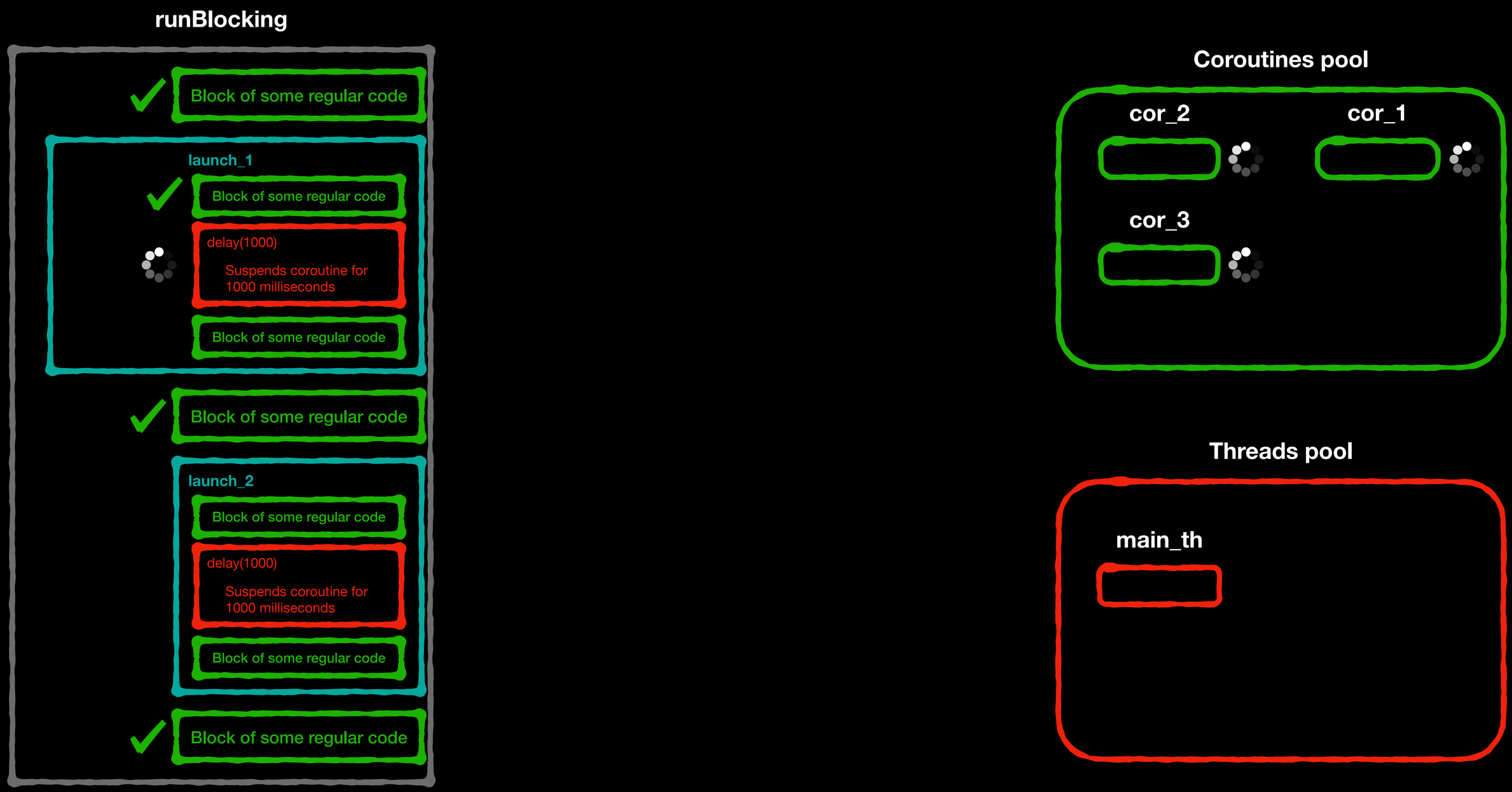
Structured concurrency - animation



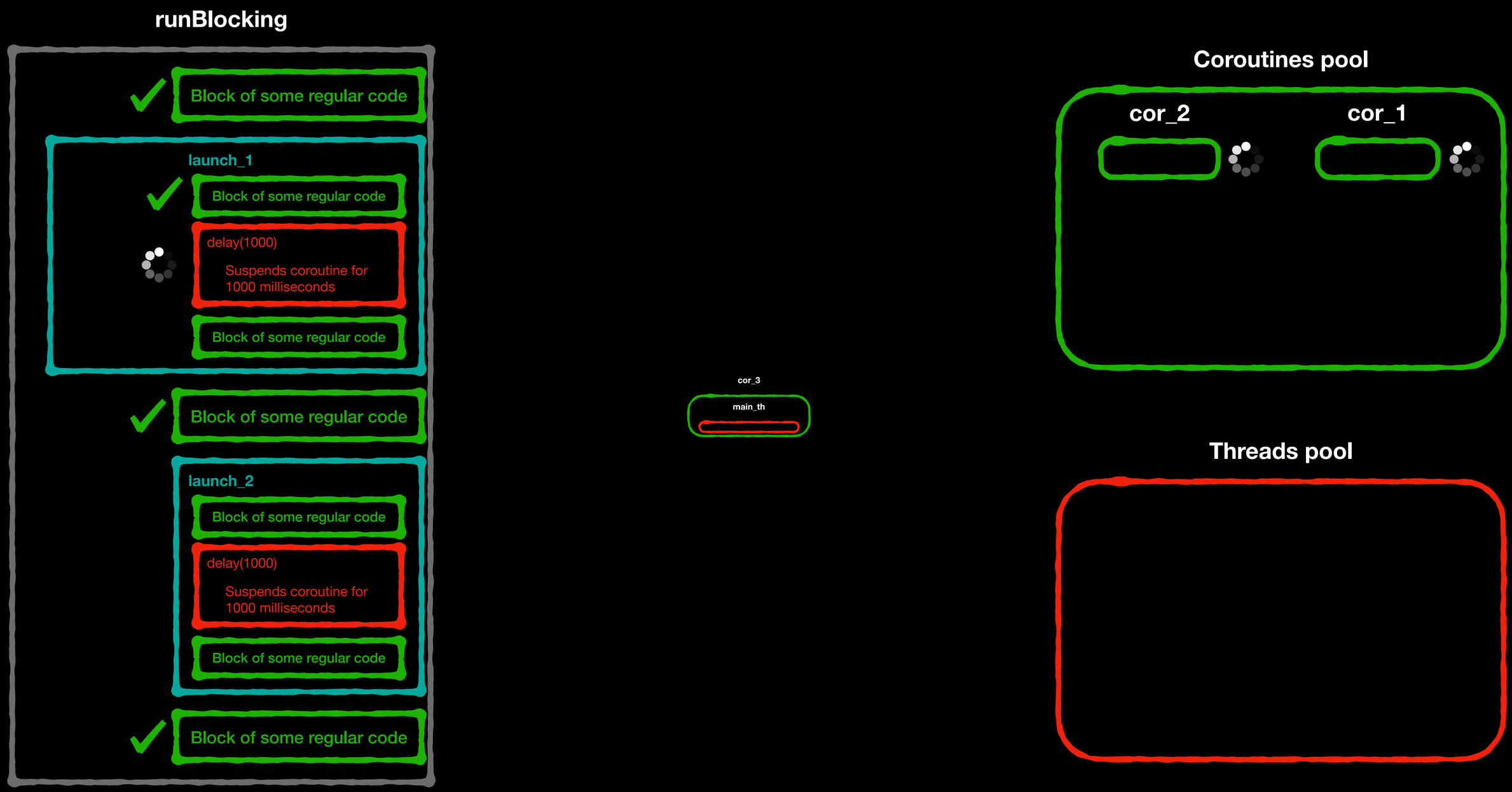




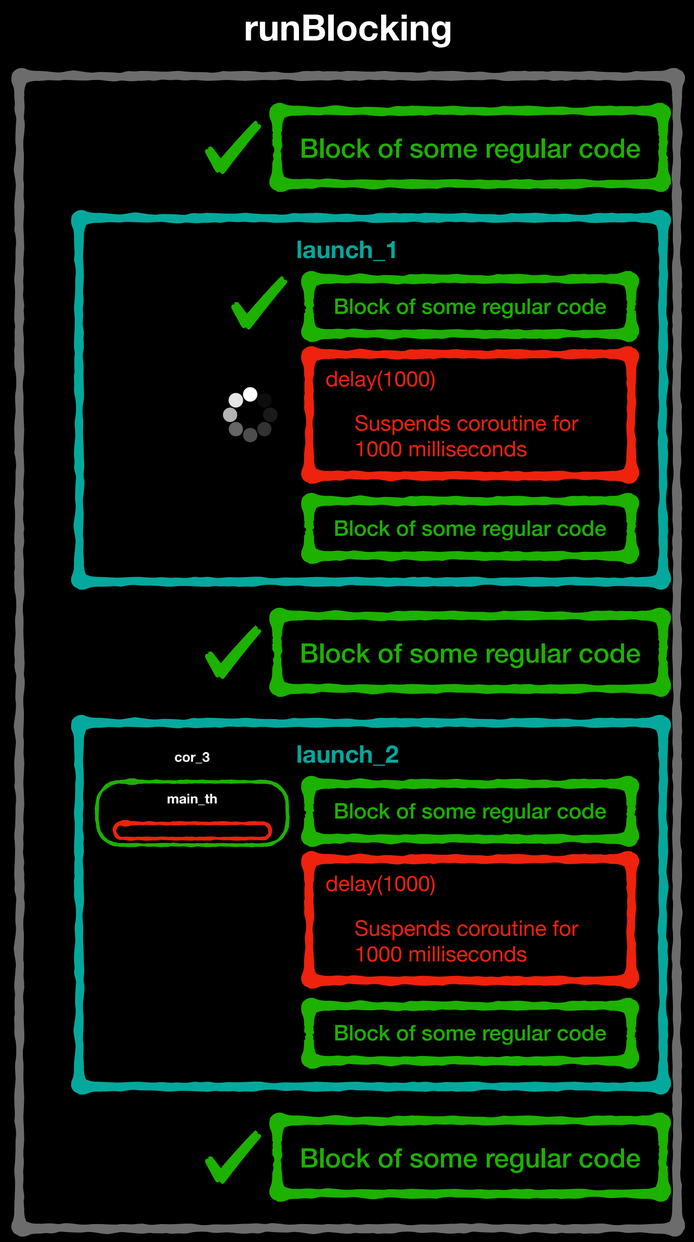
Structured concurrency - animation

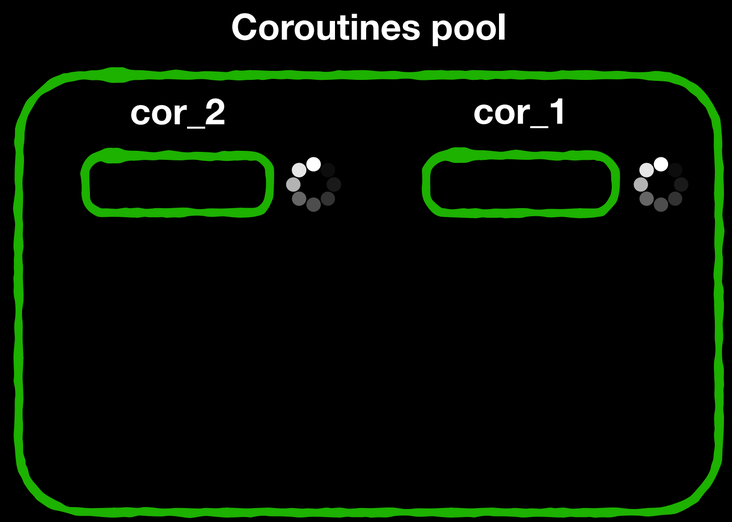


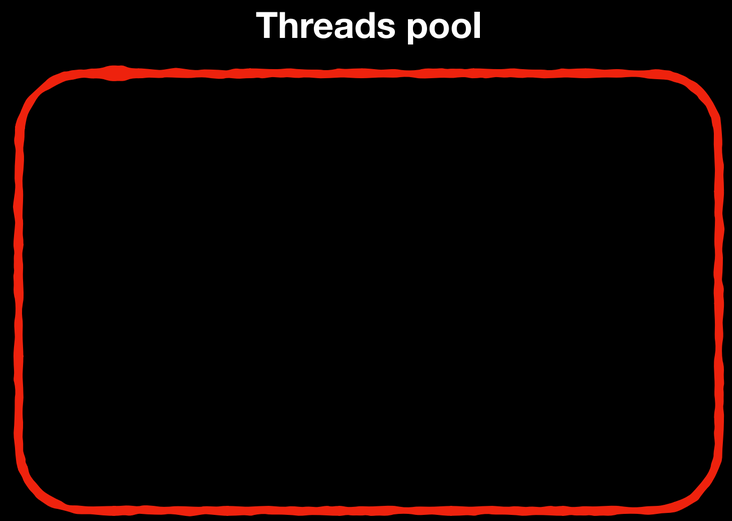
Structured concurrency - animation



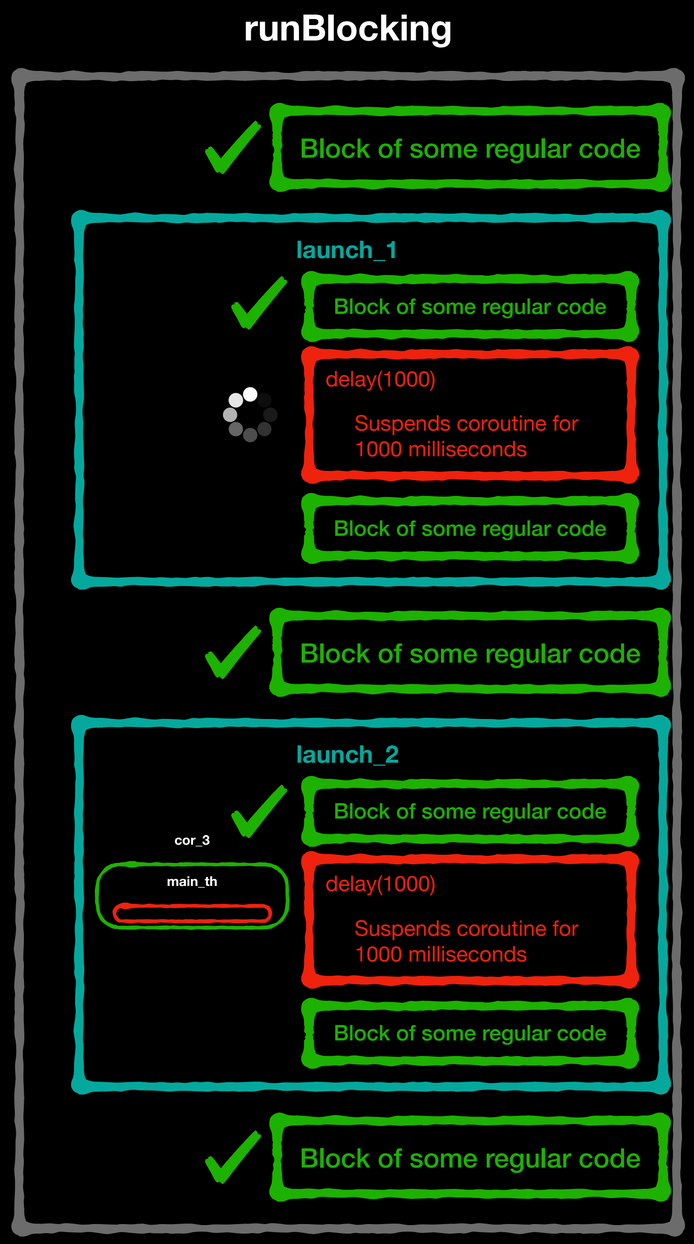
Structured concurrency - animation

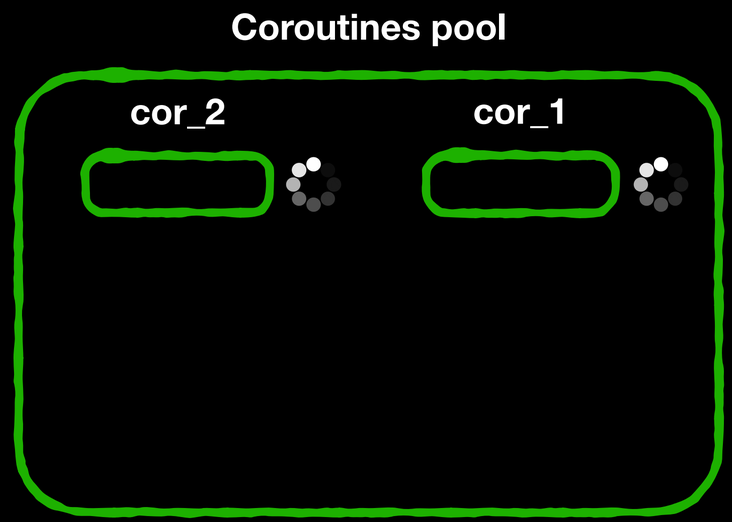


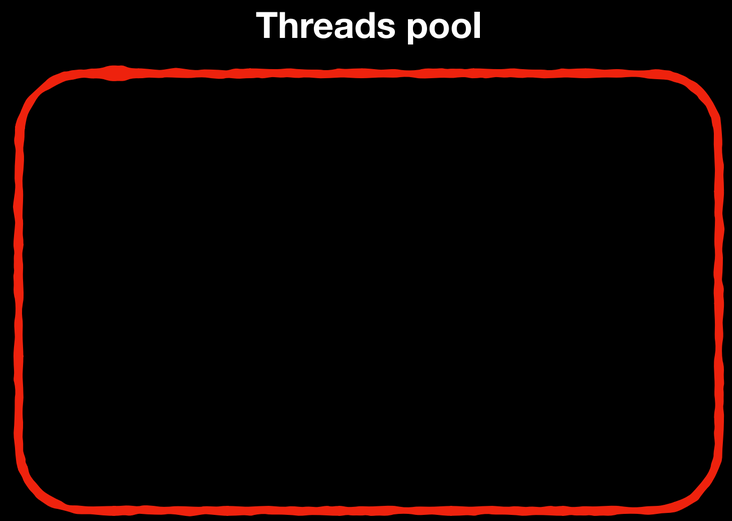




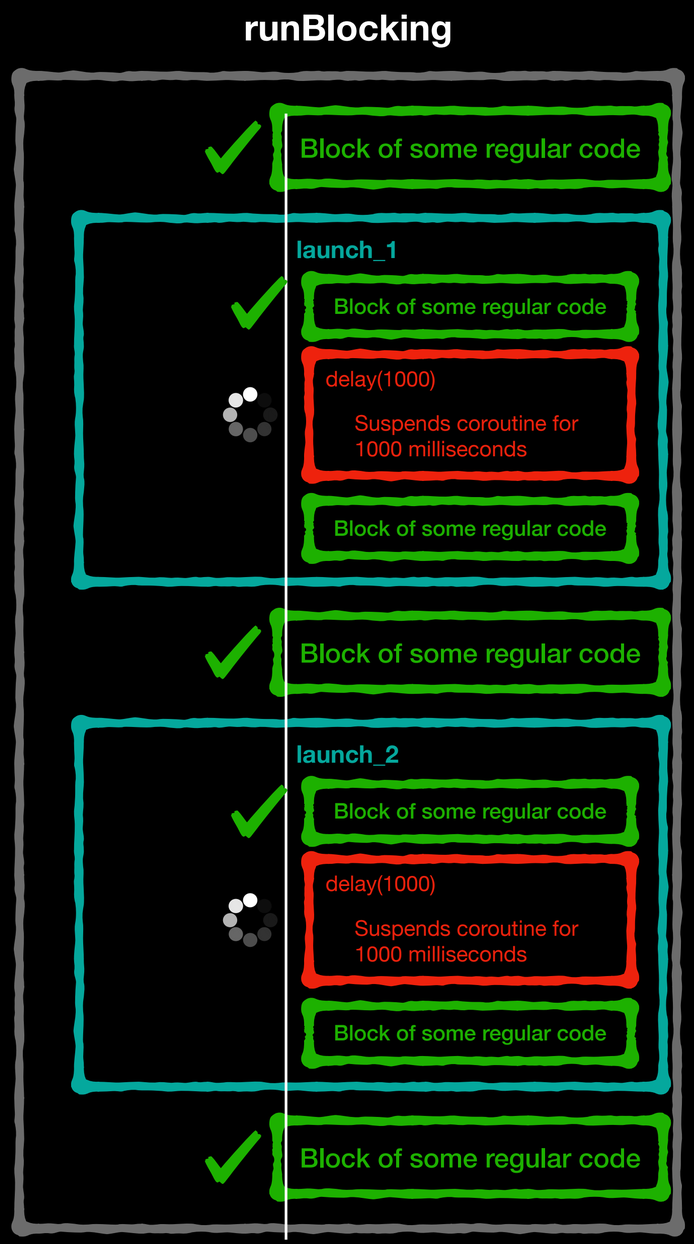
Structured concurrency - animation

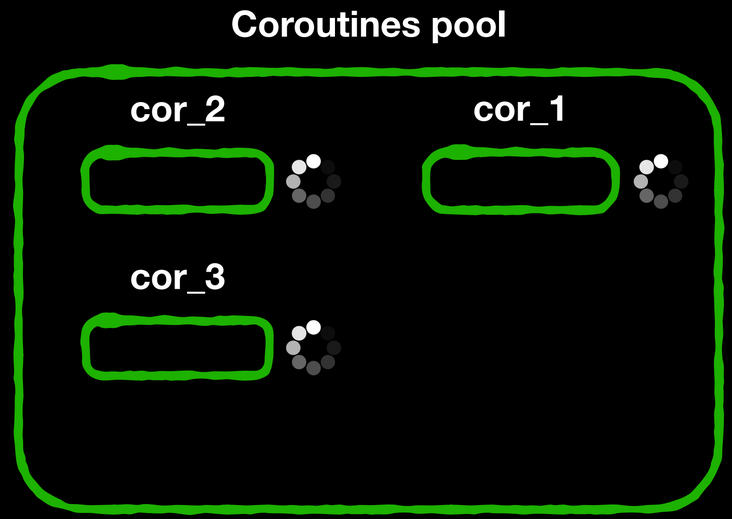




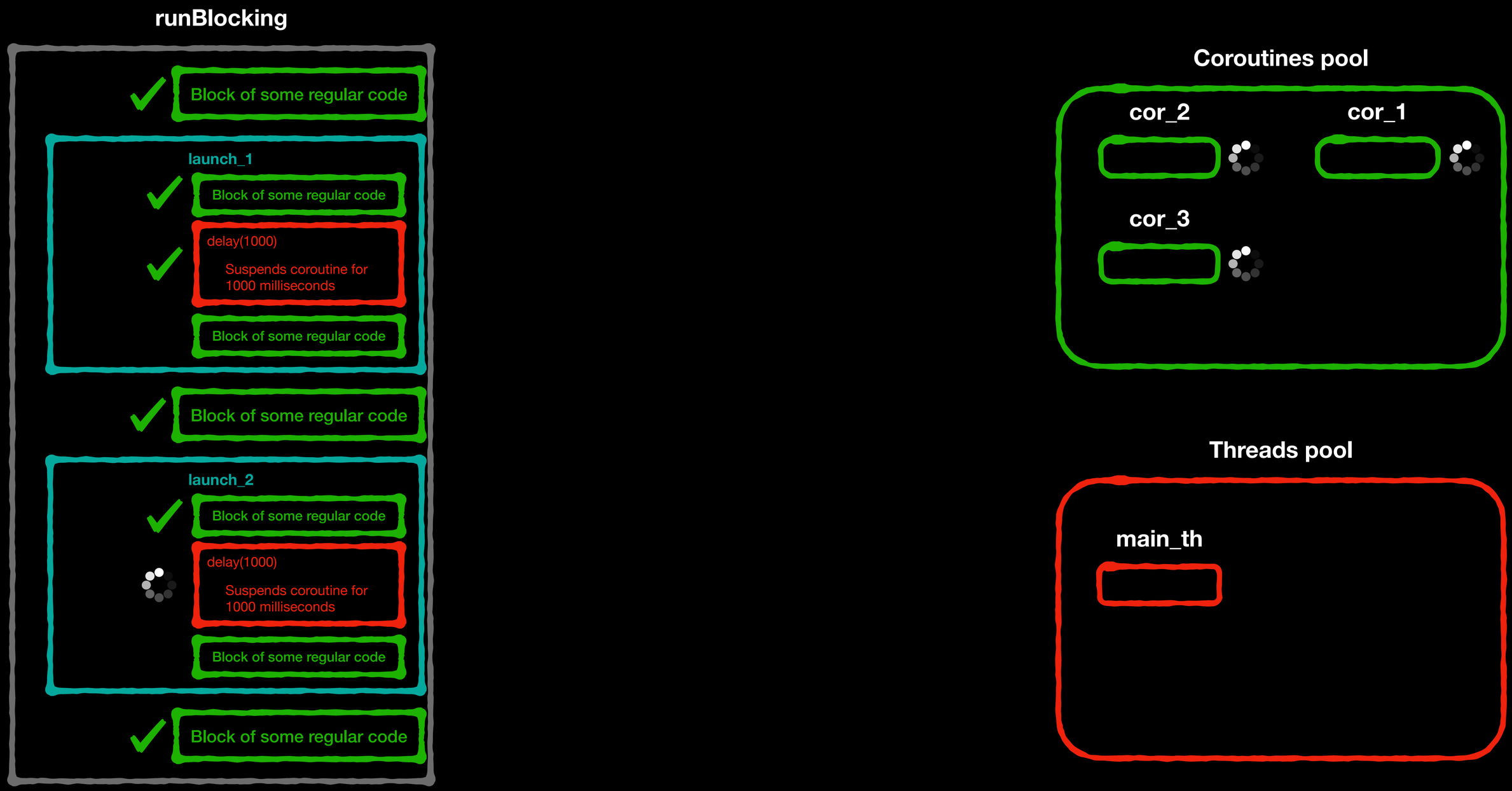


Structured concurrency - animation

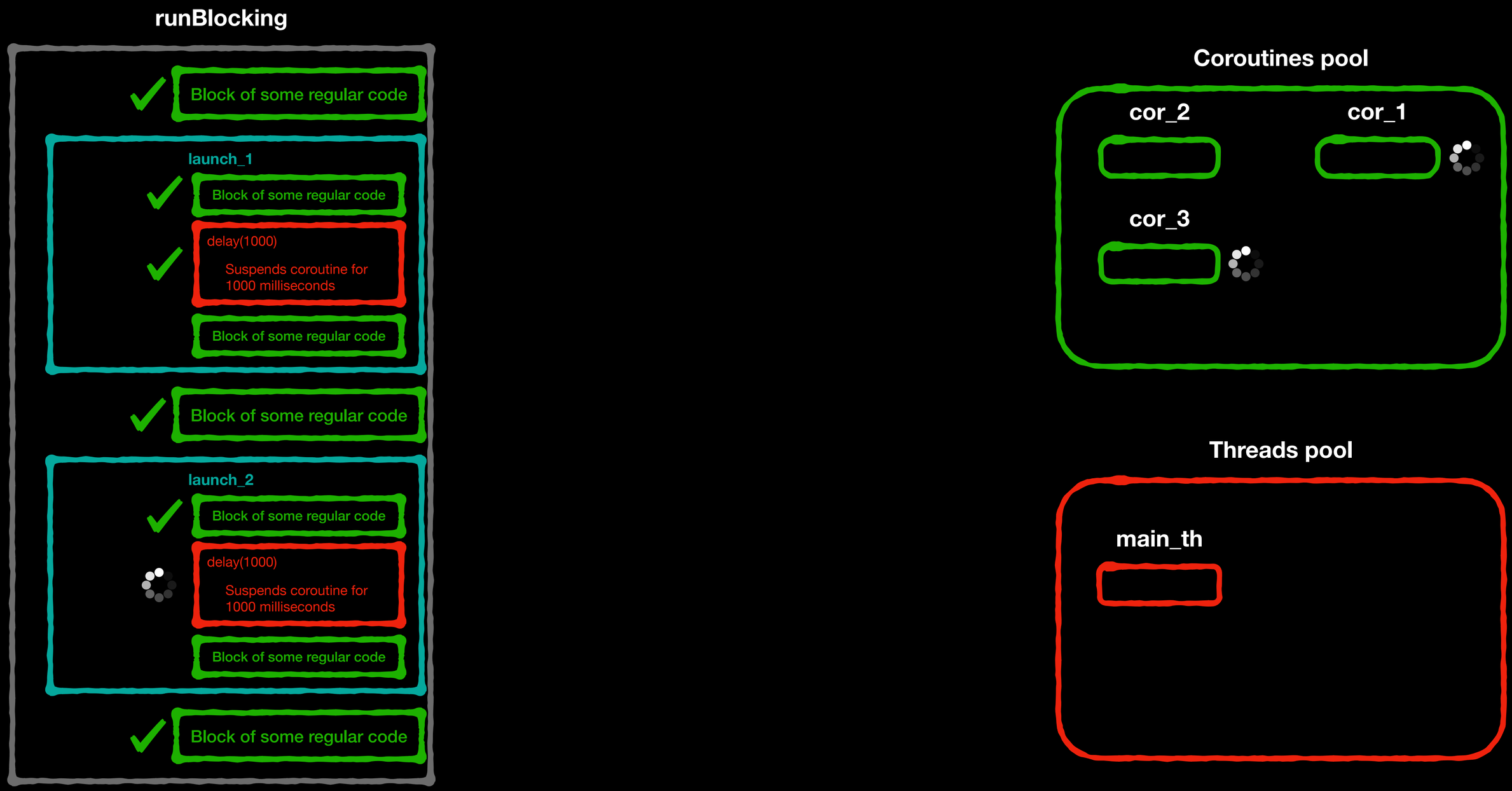




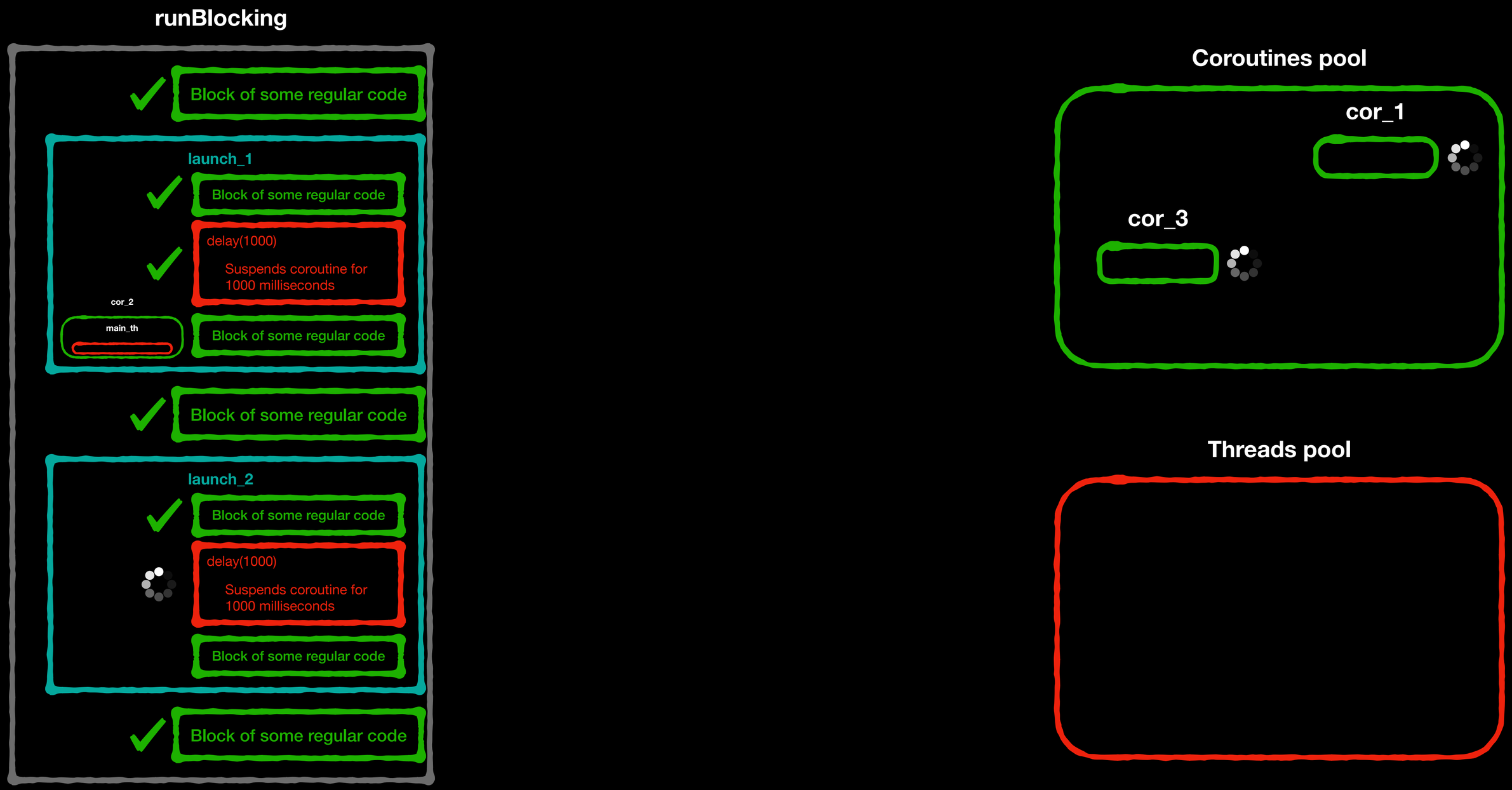
Structured concurrency - animation



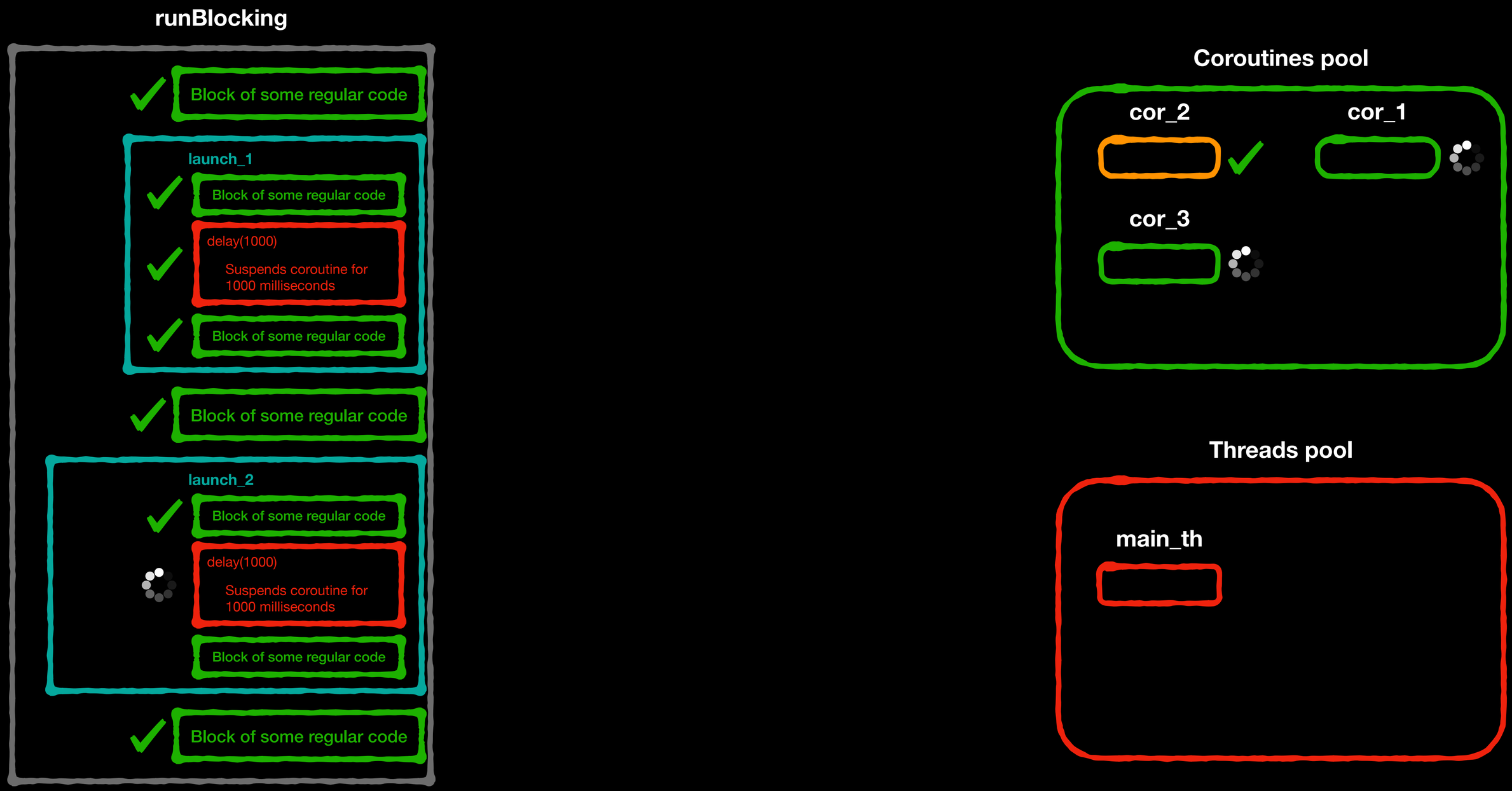
Structured concurrency - animation



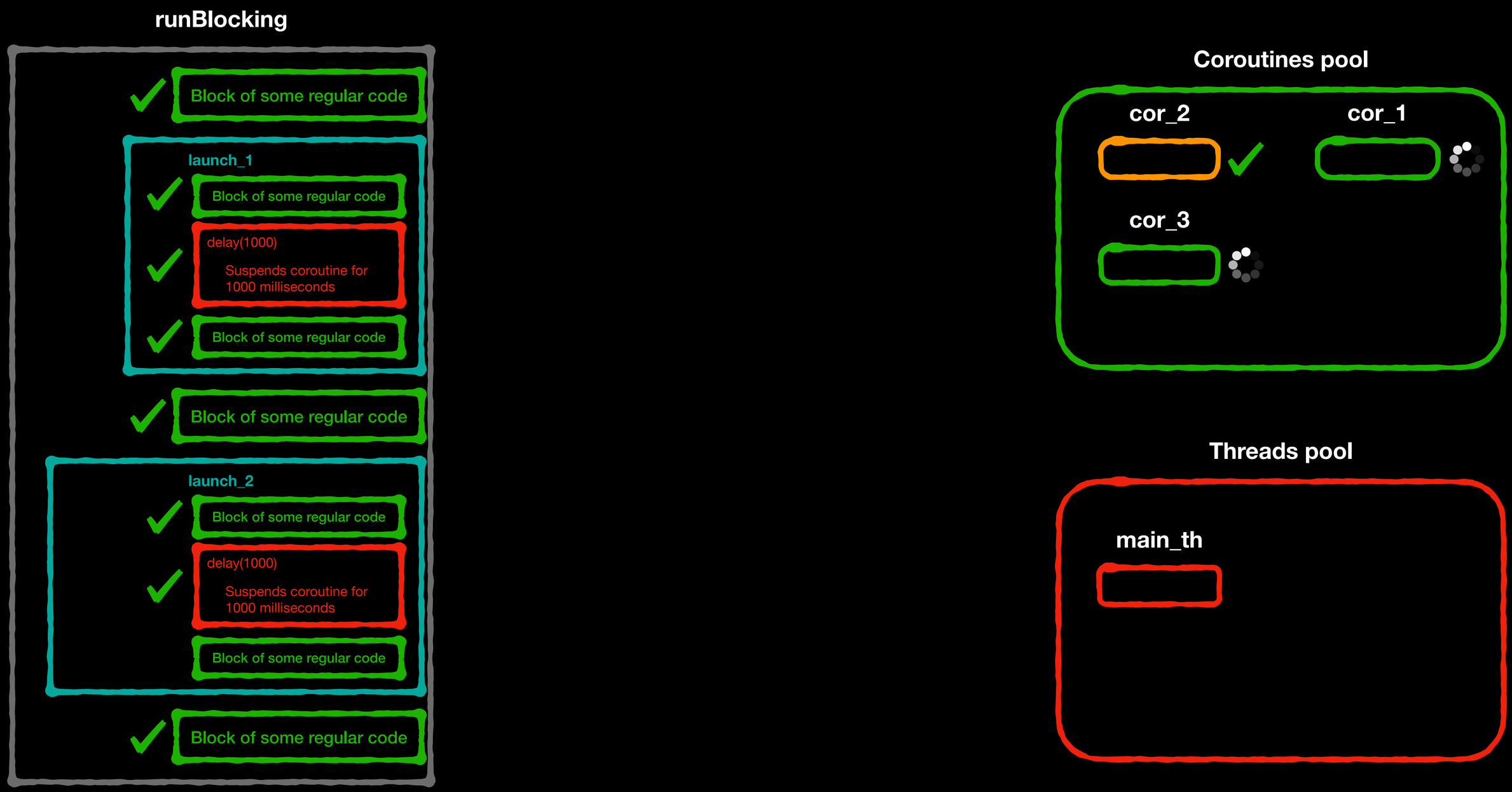
Structured concurrency - animation



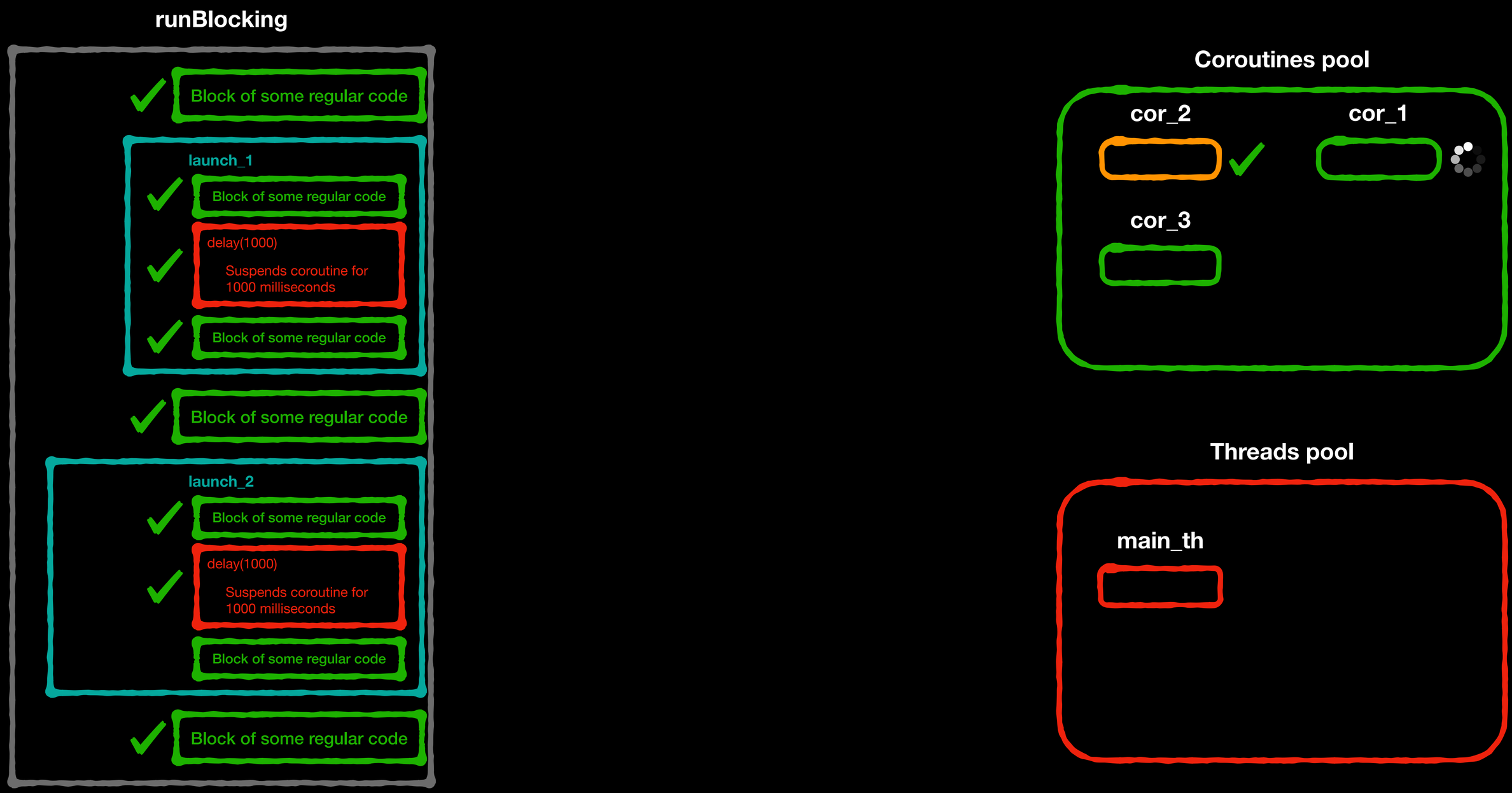
Structured concurrency - animation



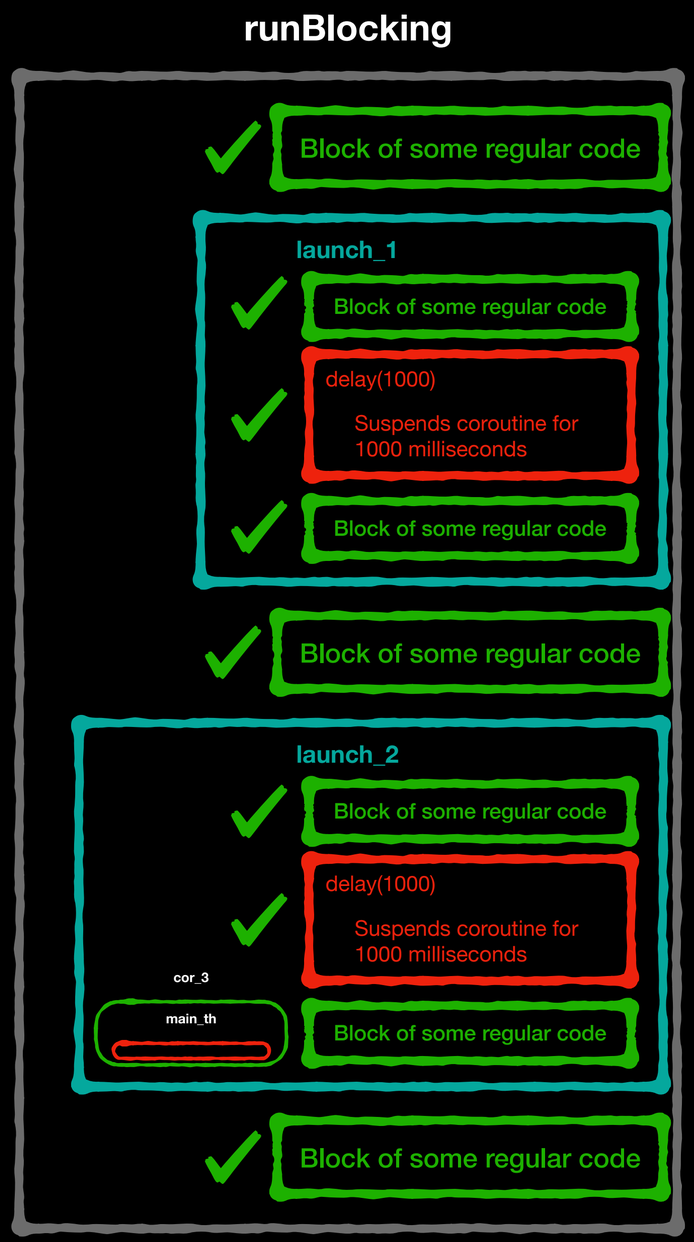
Structured concurrency - animation

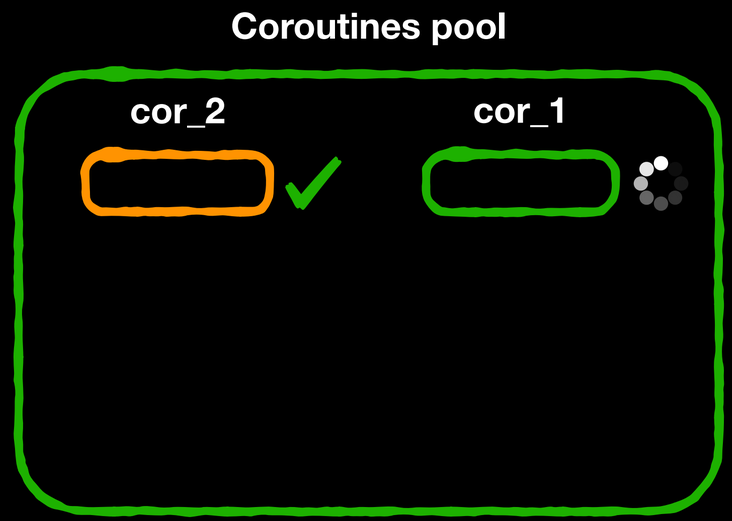


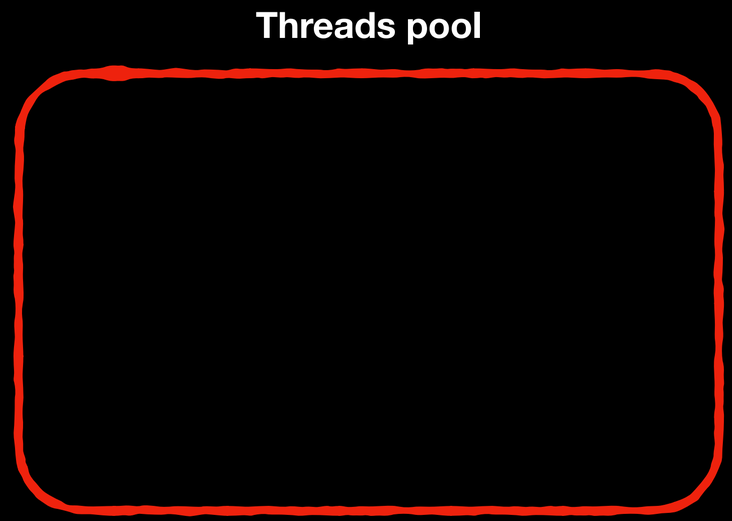
Structured concurrency - animation



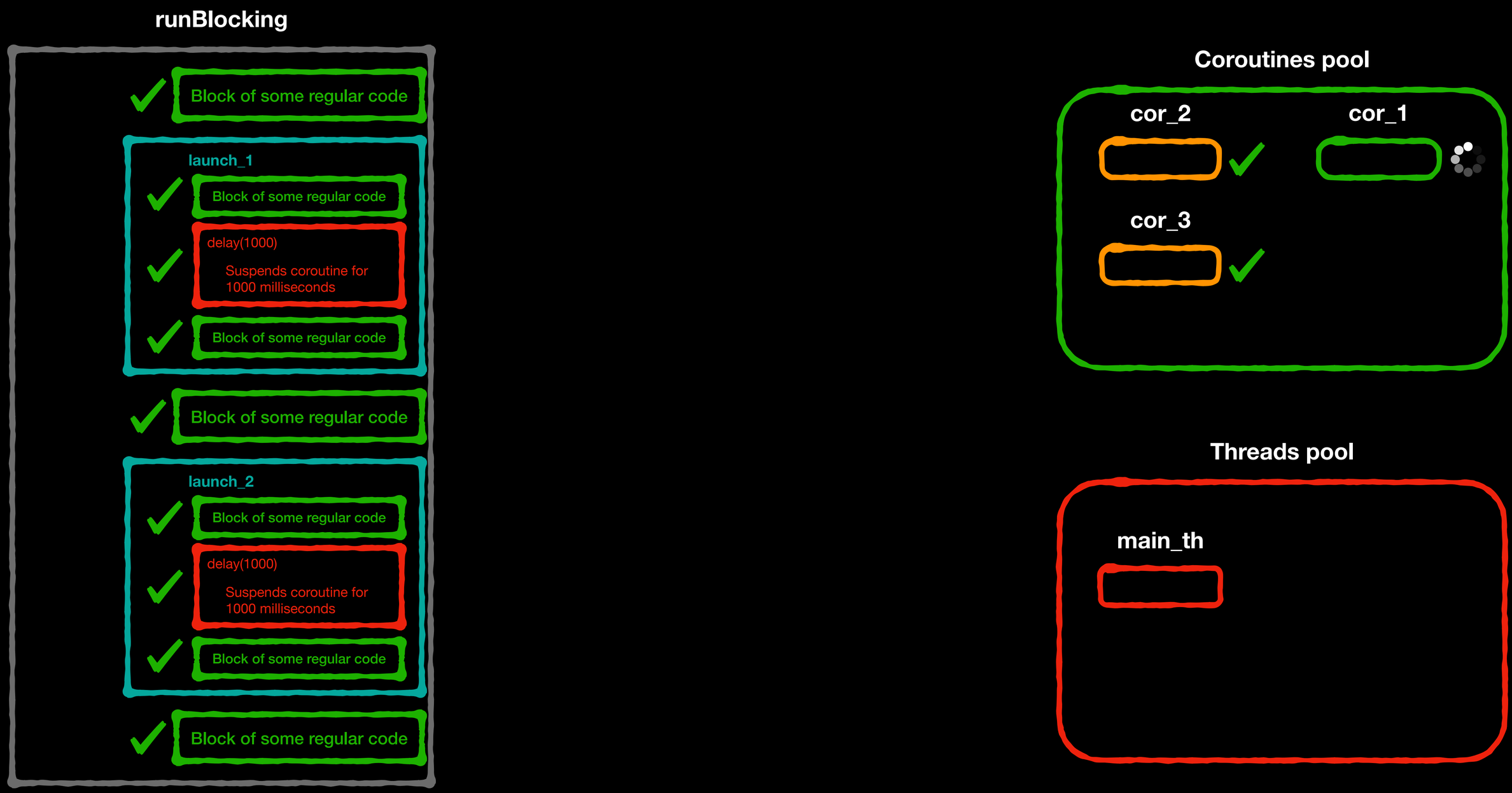
Structured concurrency - animation



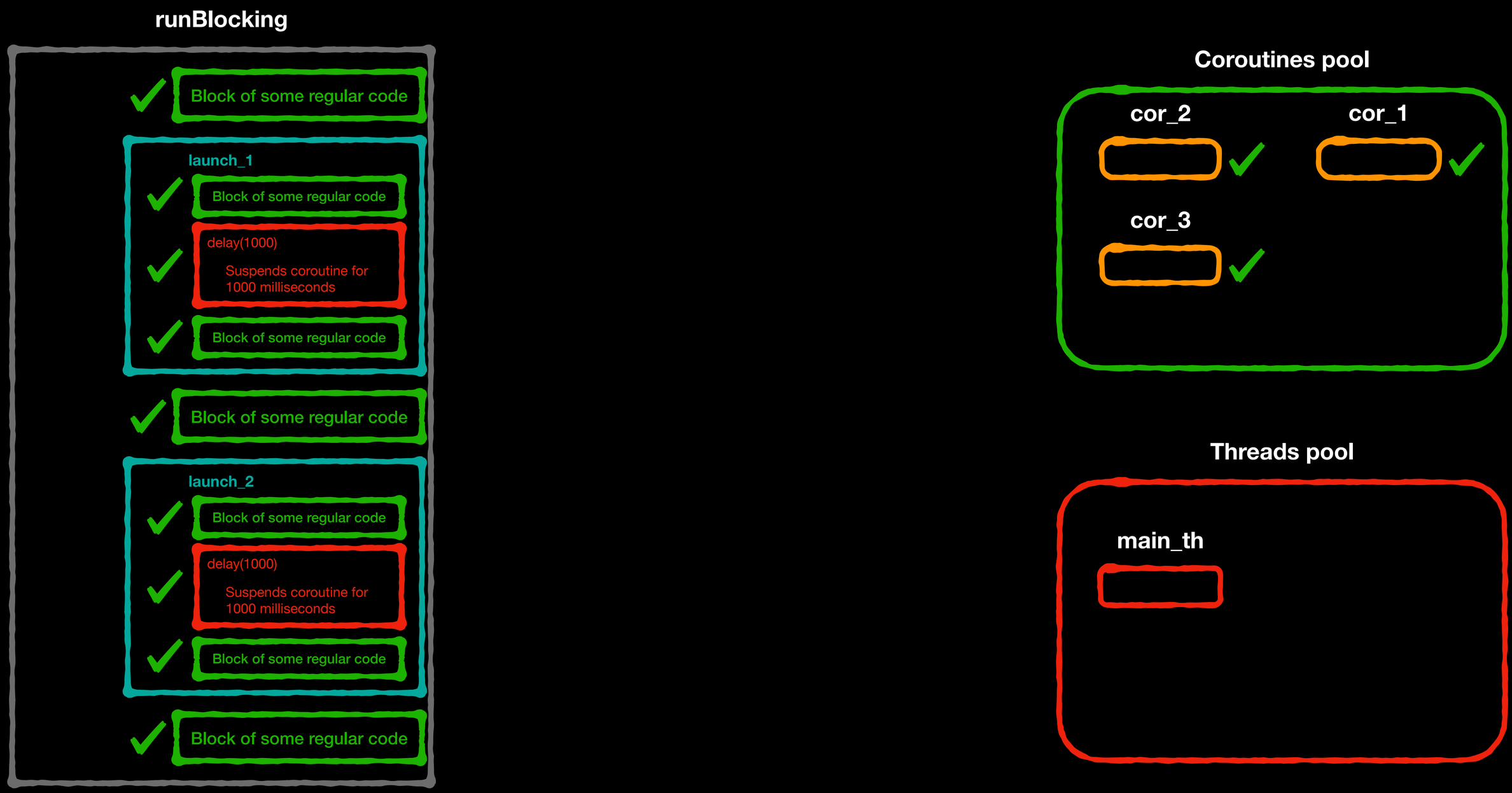




Structured concurrency - animation



Structured concurrency - animation



Coroutine definition

“A coroutine is an instance of a suspendable computation. It is conceptually similar to a thread, in the sense that it takes a block of code to run that works concurrently with the rest of the code. However, a coroutine is not bound to any particular thread. It may suspend its execution in one thread and resume in another one.”

CoroutineContext

CoroutineContext - is a collection (set) of unique coroutine configurations. Every element is separate instance of configuration.

CoroutineContext:

- inherited by coroutine from its parents
- can be overridden
- passed by suspending function inside
- coroutineContext is available inside every suspending function
- Analog of ThreadLocal in coroutine context

Example context implementation:

- Job
- Dispatchers
- CoroutineExceptionHandler
- coroutine context → defaultContext + parentContext + childContext

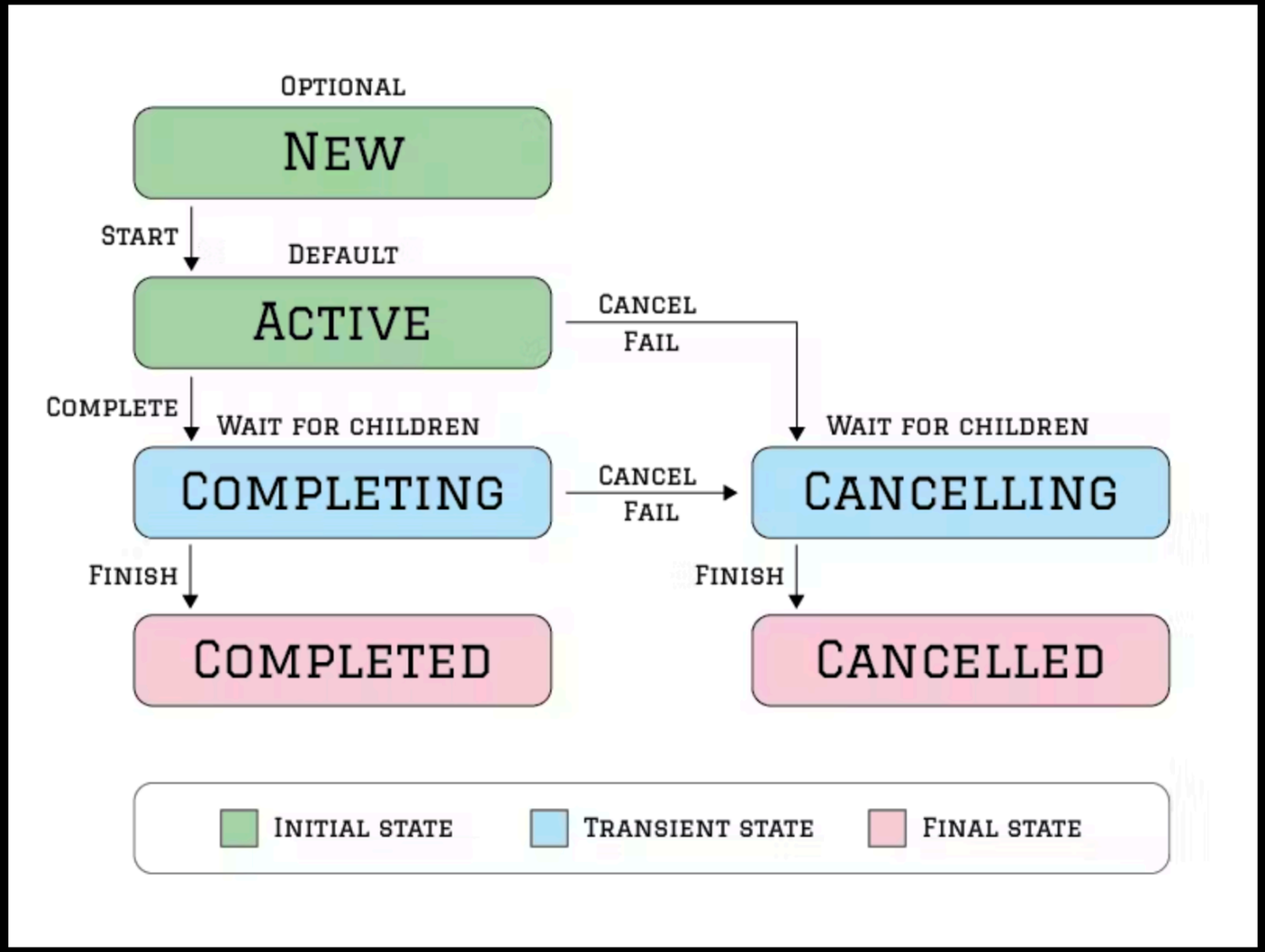
CoroutineContext - Job

Job is an implementation of CoroutineContext that responsible for handling coroutine lifecycle.

Job is the only context that is **not** inherited.

Every coroutine has it’s own Job instance that has reference to the parent of current coroutine and all children of current coroutine

CoroutineContext - Job



Source: https://maxkim.eu/things-every-kotlin-developer-should-know-about-coroutines-part-4-exception-handling

Cancellation

- Cancelled coroutine ends on first suspension point.
- Children of suspend coroutine also cancelled.
- Parent coroutine is not effected.
- Coroutine is cancelled by throwing CancellationException

To clean up:

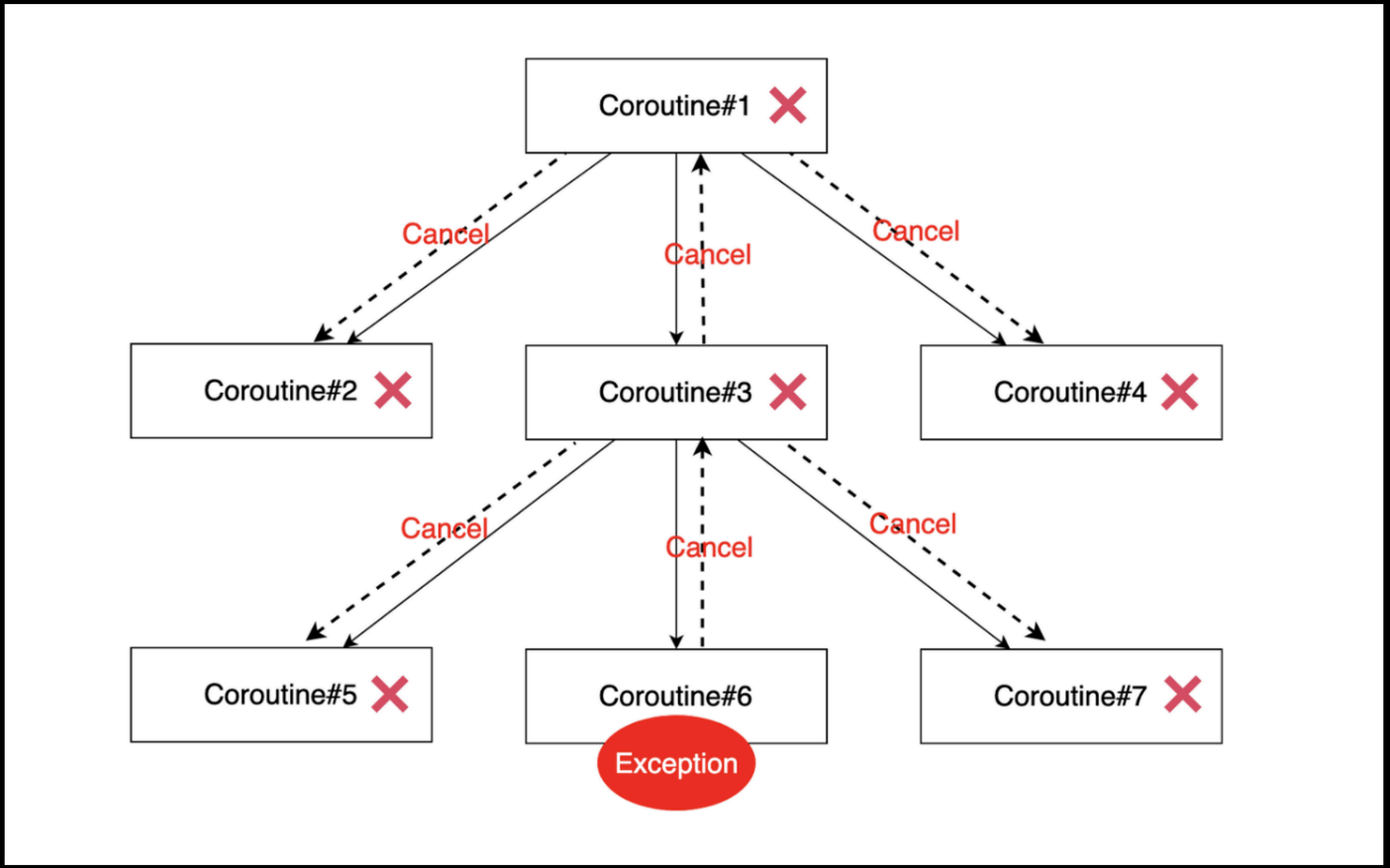
- invokeOnCompletion
- suspendCancellableCoroutine.invokeOnCancellation
- ensureActive, yield

Exception handling

Broken coroutine:

- Cancel all its children
- Propagates error to parent coroutine

Exception handling



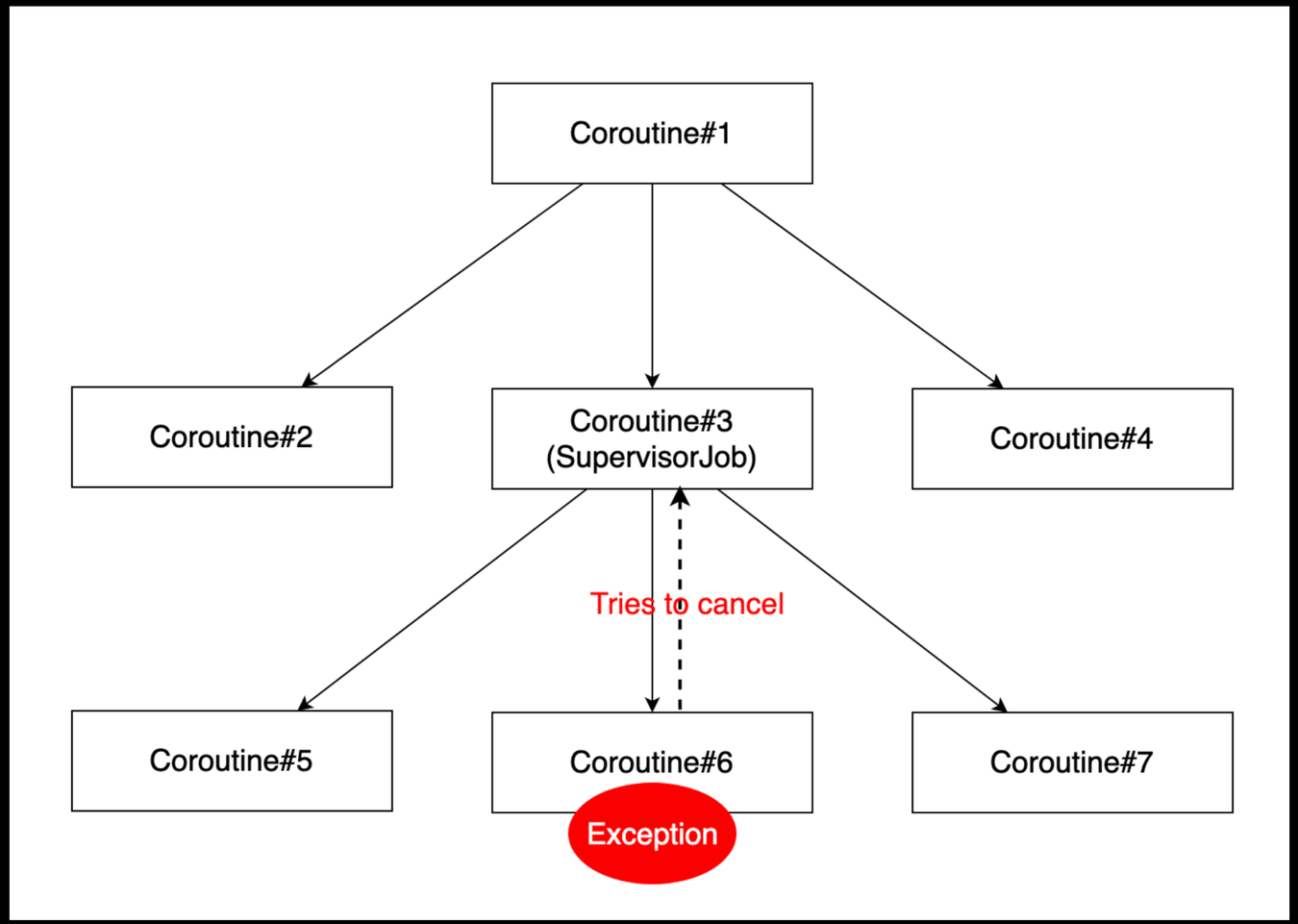
Source: https://icarus8050.tistory.com/151

Exception handling

To stop breaking

- use CancellationException
- SuperviserJob
- CoroutineExceptionHandler
- superviserScope
- coroutineScope

Exception handling



Source: https://icarus8050.tistory.com/151

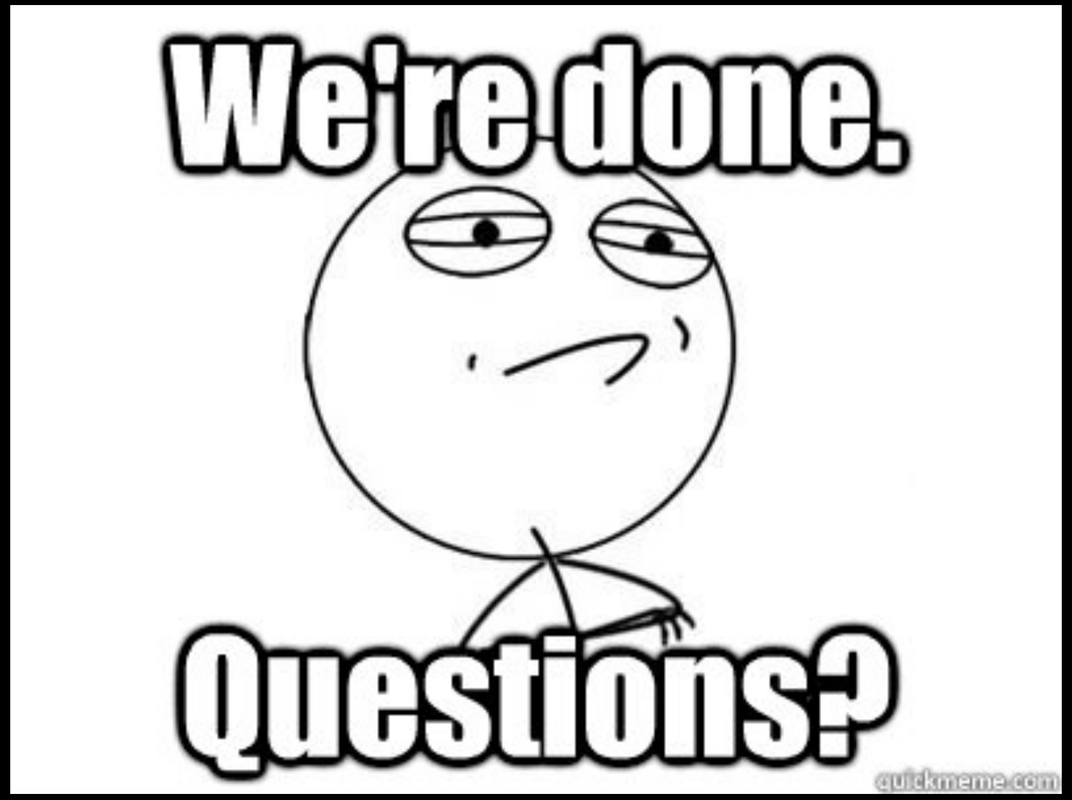
Coroutine scope functions

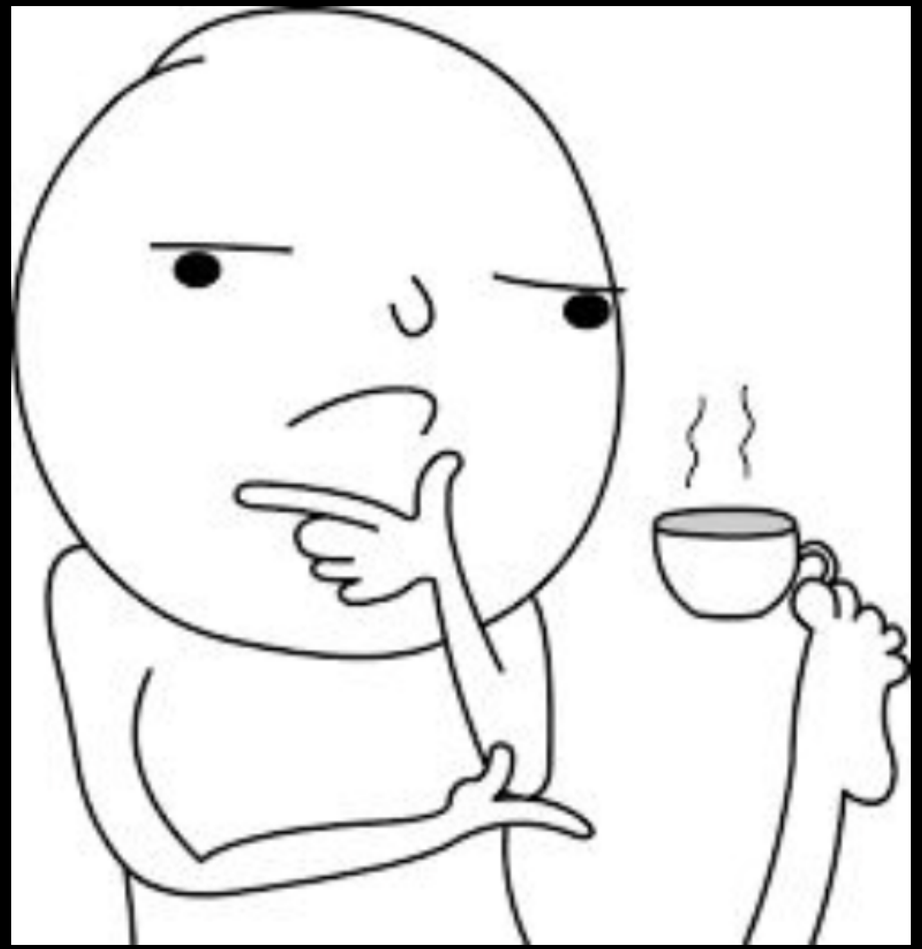
Coroutine scope functions:

- create an environment where you can launch one or more coroutines and ensure they don't linger on after the block completes.
- provide CoroutineScope inside suspended function that inherits CoroutineContext from it’s parent

Examples:

- coroutineScope
- supervisorScope
- withContext
- withTimeout


We're done.
Questions?
quickmeme.com



Thank you for your attention!