Swift 5.5
20 Sept 2021

- Async/ await SE-0296
- Async let SE-0317
- Async/ await / async let interoperability with ObjC SE-0297
- Interfacing async with callbacks SE-0300
- Async Sequences SE-0298
- AsyncStreams SE-0314
- Structured concurrency SE-0304
- Tasks local values SE-0311
- Actors SE-0306
- Actor isolation control SE-0313
- Global Actors SE-0316

Swift 5.5.2
Dec 14, 2021

- Clarification of the semantics of Task where the main function runs SE-0323

Swift 5.6
Mar 14, 2022

- Remove sendable conformance in unsafe pointers SE-0331
- Support incremental migration of concurrency checks SE-0337

Swift 5.7
Sep 12, 2022

- Sendable + closures SE-0302
- Distributed Actors SE-0336
- Clarify how non-isolated async Actor functions should work SE-0338
- NoAsync - mark async unavailable SE-0340
- Detailing how concurrency work on top level code SE-0343
- Specifying how the runtime should work for distributed actors SE-0344

Swift 5.9
Sep 18, 2023

- Discarding task groups. Improving limitations of SE-304 SE-0381
- AsyncStream factory. Improving SE-0314 SE-0388
- Custom actor executor SE-0392

Swift 5.10
Mar 5, 2024

- Specifying how init and deinit should work on actors SE-0327
- Isolation for default property values and default parameters values. SE-0411
- Strict concurrent for global variables SE-0412

Swift 6.0
Sep 17, 2024

- Regions based isolation SE-0414
- Task executor preferences SE-0417
- Inferring sendable for keyPaths and methods SE-0418
- Changes in concurrency rules for global actor isolated types SE-0434
- AssumeIsolated / Dynamic isolation checks for non-strict concurrency code SE-0423
- Custom isolation checks for SerialExecutor SE-0424
- "sending" parameters and results values SE-0430
- Giving async functions actor isolation / inheritance of actor isolation SE-0420
- isolated(any) function type SE-0431

Swift 6.1

- Isolated sync deinit for actors and global actors. Lifting restriction from SE-0327 SE-0371
- Inferring TaskGroup child results SE-0442
- Allow preventing global-actor inferences, using nonisolated SE-0449

async await SE-0296

works at the compiler level, it translates asyn/await into closures

was the foundation for the other concurrency features

can not be called from SYNC functions (since those do not abandon their threads)

async functions are able to give up their thread's stack and use their own storage

Asynchronous functions should avoid calling functions that can \ block the thread

many async functions are part of an actor. Swift guarantees that they return to their actor after suspension

deinit and storage accessors (i.e., the setters for properties and subscripts) cannot be async.

when they resume they might be using another thread

```
class Teacher {

    deinit async {}                                   Deinitializers cannot have a name

    var schoolDirectorName: String {
        set async {}                                  2  'set' accessor cannot have specifier 'async'
        get async { "name" }
    }

    subscript(index: Int) async -> Int {              3  Cannot find type 'async' in scope
        get {
            // Return an appropriate subscript value here.
        }
        set(newValue) {
            // Perform a suitable setting action here.
        }
    }

    init(from schoolName: String) async throws {
        // initialize
    }

    private func raiseHand() async -> Bool {
        true
    }
}
```

suspension interrupt atomicity, the async function can leave its context, meanwhile the context can be used to execute other code

await can not be used in defer

if overload resolution is needed for a function with an async and a sync version: prefers non-async functions within a synchronous context /
prefers async functions within an asynchronous context

protocols: an async requirement can be satisfied by either async or sync, a sync requirement can not be satisfied by async.

there is an implicit conversion from a synchronous function type to its corresponding asynchronous function type

objC interoperability SE-0297

Swift provides bridging from ObjC to async await

methods in ObjC are automatically converted to async when they fullfill some conditions (See proposal SE-0297)

functions in Swift can be marked with @objc and the compiler will translate them into ObjC callbacks, when possible

an actor can not inherit from a class, with the exception of NSObject, to precisely provide bridging.

An actor function can be marked @objc only if it is async or is non isolated. ObjectiveC does not know how to provide actor isolation of sync functions

markers were added to help the compiler and annotate old ObjC code with for example swint_async and swift_async_error (see SE-0297 for the whole list of attributes)

async let SE-0317

was introduced as a way to hide the complexities of setting up child tasks and waiting for their results

the "let" indicates that it is a local constant the "async" part mean that it is evaluated in a concurrent child task

By default, child tasks use the **global, width-limited, concurrent executor**

the child task closure created is sendable and non-isolated

It is illegal to declare an async var

if you do not wait for an async let, the function still will take as much time to return as the longest of its child tasks takes to complete

Special attention needs to be given to the async let _ = ...it will run and be cancelled and awaited-on implicitly, as the scope it was declared in is about to exit

It is *not* legal to escape a async let value to an escaping closure, because structures backing the async let implementation may be allocated on the stack rather than the heap. This makes them very efficient, however, make it unsafe to pass them to any escaping contexts

Limitation: the number of child-tasks is *dynamic,* it is not possible to express using async let because we'd have to know how many async let declarations to create at *compile time*

Limitation: Because async let declarations must be awaited on it is not possible to express "whichever completes first", and a task group must be used to implement such API

```swift
func makeDinner() async throws -> Meal {
  async let veggies = chopVegetables()
  async let meat = marinateMeat()
  async let oven = preheatOven(temperature: 350)

  let dish = Dish(ingredients: await [try veggies, meat])
  return try await oven.cook(dish, duration: .hours(3))
}
```

closure + async await SE-0300

continuations are a way to mix callback based code with async code

swift provides continuation APIs, which are given to async functions. Inside closure based code can be called

the closure based code runs in the current task's context

when the closure based code finished the continuation must be resumed with success or error

resume must be called ONLY ONCE

if the continuation is unsafe, and the resume is called more than once, Swift produces undefined behaviour

it is better to use checked continuations, since it is a wrapper on top of unsafe, which checks how many times the resume is called, and also check for forgotten resumes, plus trap at runtime if there is any error in the handling of the continuation

```
func buyVegetables(
  shoppingList: [String],
  // a) if all veggies were in store, this is invoked *exactly-once*
  onGotAllVegetables: ([Vegetable]) -> (),

  // b) if not all veggies were in store, invoked one by one *one or more times*
  onGotVegetable: (Vegetable) -> (),
  // b) if at least one onGotVegetable was called *exactly-once*
  //    this is invoked once no more veggies will be emitted
  onNoMoreVegetables: () -> (),

  // c) if no veggies _at all_ were available, this is invoked *exactly once*
  onNoVegetablesInStore: (Error) -> ()
)
// returns 1 or more vegetables or throws an error
func buyVegetables(shoppingList: [String]) async throws -> [Vegetable] {
  try await withUnsafeThrowingContinuation { continuation in
    var veggies: [Vegetable] = []

    buyVegetables(
      shoppingList: shoppingList,
      onGotAllVegetables: { veggies in continuation.resume(returning: veggies) },
      onGotVegetable: { v in veggies.append(v) },
      onNoMoreVegetables: { continuation.resume(returning: veggies) },
      onNoVegetablesInStore: { error in continuation.resume(throwing: error) },
    )
  }
}

let veggies = try await buyVegetables(shoppingList: ["onion", "bell pepper"])
```

```
// returns 1 or more vegetables or throws an error
func buyVegetables(shoppingList: [String]) async throws -> [Vegetable] {
  try await withCheckedThrowingContinuation { continuation in
    var veggies: [Vegetable] = []

    buyVegetables(
      shoppingList: shoppingList,
      onGotAllVegetables: { veggies in continuation.resume(returning: veggies) },
      onGotVegetable: { v in veggies.append(v) },
      onNoMoreVegetables: { continuation.resume(returning: veggies) },
      onNoVegetablesInStore: { error in continuation.resume(throwing: error) },
    )
  }
}
```

Async sequences SE-0298

async/await functions return 1 value... async/await sequences are the provided Swift way, to return many overtime

the main point is been able to return more than one value, along the time, without waiting for all the values to be calculated or read

the 2 protocols for sequences, provided by the standard lib are: AsyncSequence and AsyncIteratorProtocol

the compiler produces code to support using for...in with async sequences

The AsyncIterator MUST check for Task isCancelled and decide if to return the partially calculated result or if to return nil

After an AsyncIteratorProtocol types returns nil or throws an error from its next() method, all future calls to next() must return nil

```swift
public protocol AsyncSequence {
  associatedtype AsyncIterator: AsyncIteratorProtocol where AsyncIterator.Element == Element
  associatedtype Element
  __consuming func makeAsyncIterator() -> AsyncIterator
}

public protocol AsyncIteratorProtocol {
  associatedtype Element
  mutating func next() async throws -> Element?
}
```

async sequences come with some already defined cool functions

Function	Note
`contains(_ value: Element) async rethrows -> Bool`	Requires `Equatable` element
`contains(where: (Element) async throws -> Bool) async rethrows -> Bool`	The `async` on the closure allows optional async behavior, but does not require it
`allSatisfy(_ predicate: (Element) async throws -> Bool) async rethrows -> Bool`	
`first(where: (Element) async throws -> Bool) async rethrows -> Element?`	
`min() async rethrows -> Element?`	Requires `Comparable` element
`min(by: (Element, Element) async throws -> Bool) async rethrows -> Element?`	
`max() async rethrows -> Element?`	Requires `Comparable` element
`max(by: (Element, Element) async throws -> Bool) async rethrows -> Element?`	
`reduce<T>(_ initialResult: T, _ nextPartialResult: (T, Element) async throws -> T) async rethrows -> T`	
`reduce<T>(into initialResult: T, _ updateAccumulatingResult: (inout T, Element) async throws -> ()) async rethrows -> T`	

Async streams SE-0314

AsyncStream and AsyncThrowingStream were added to help starting to write more async sequences

Continuations were added to support connecting callback based code to async function. BUT this only worked for 1 single value returned... so something new was needed for closure based code which returns many values over time

when creating an stream, you receive a continuation, but this one can be called multiple time to YIELD elements

elements yielded are accumulated and buffered, until a consumer receives then

a continuation can also be finished. Then any accumulated value is given to the consumer before terminating with nil or throwing an exception

```
func buyVegetables(
  shoppingList: [String],

  // a) invoked once for each vegetable in the shopping list
  onGotVegetable: (Vegetable) -> Void,

  // b) invoked once all available veggies have been retrieved
  onAllVegetablesFound: () -> Void,

  // c) invoked if a non-vegetable food item is encountered
  // in the shopping list
  onNonVegetable: (Error) -> Void
)

// Returns a stream of veggies
func findVegetables(shoppingList: [String]) -> AsyncThrowingStream<Vegetable> {
  AsyncThrowingStream { continuation in
    buyVegetables(
      shoppingList: shoppingList,
      onGotVegetable: { veggie in continuation.yield(veggie) },
      onAllVegetablesFound: { continuation.finish() },
      onNonVegetable: { error in continuation.finish(throwing: error) }
    )
  }
}
```

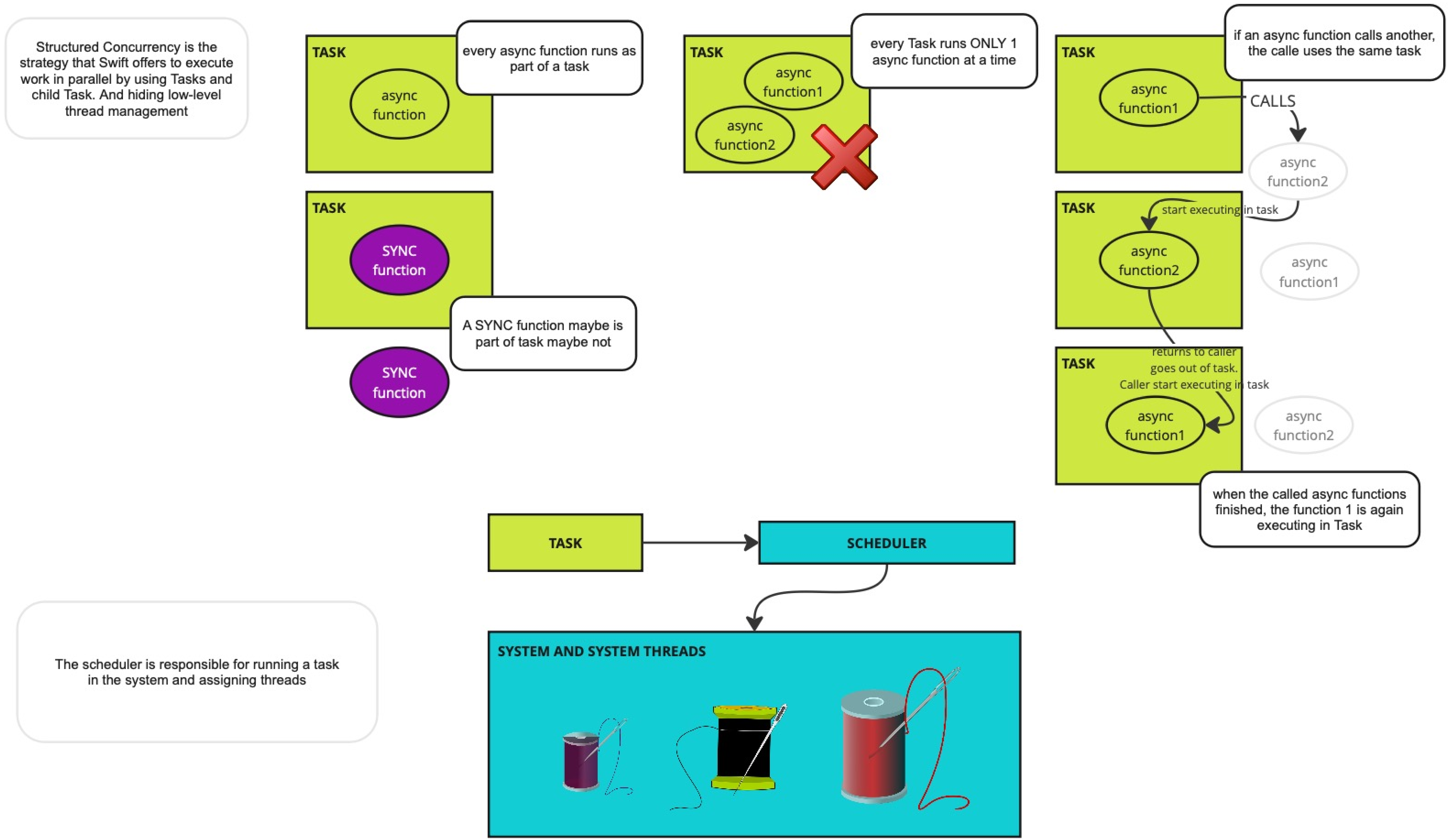
Async streams and View Controllers

When using AsyncStream, or a long running task, from a view controller life cycle, you should keep a reference to the task and **cancel it on deinit**

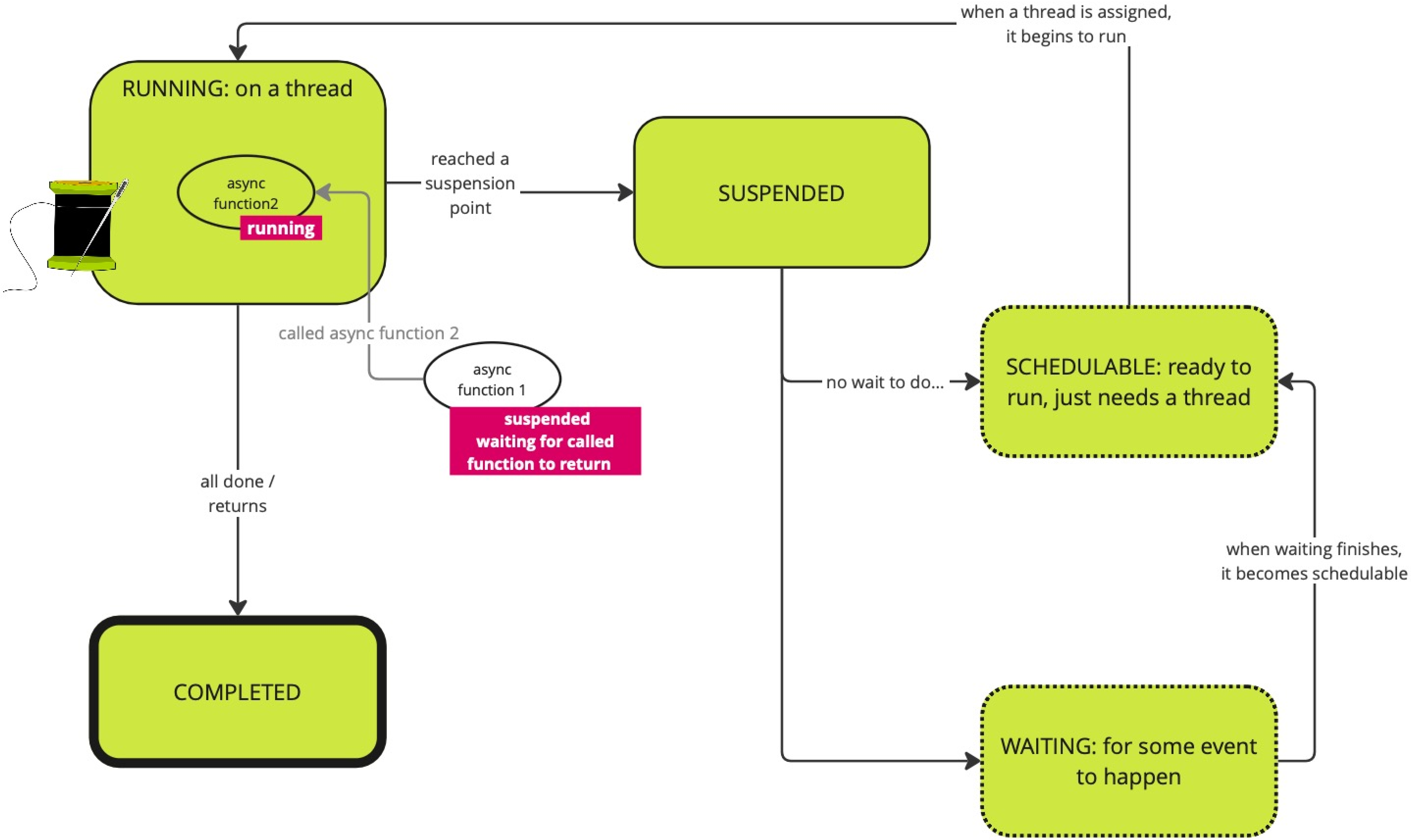
make sure to use weak self, and never capture again a strong reference inside the task, this will cause self to be strongly captured. You can use self? in the whole code inside the task to avoid memory cycle.

```swift
class AViewController: UIViewController {

    /// keep a reference if the task is super long
    /// if the task finished quickly, is not a big problem
    /// it will cancel itself when done
    /// AsyncStream -> ALWAYS cancel them
    private var loadingTask: Task<Void, Never>?
    /// just a worker to exemplify an async stream creation
    private var aWorker = AWorker()
    /// keep a reference to the task used to call the async stream... so you can cancel it
    /// when the view controller deinit - make sure not to make any reference cycle inside that task
    private var aTaskReference: Task<Void, Never>?

    deinit {
        loadingTask?.cancel()
        aTaskReference?.cancel()
    }

    override func viewDidLoad() {
        super.viewDidLoad()

        // example for a normal Task (without AsyncStream inside)
        loadingTask = Task {
            // some very long calculation
        }
        aTaskReference = Task {
            for await value in aWorker.getAsyncStream() {
                // do something with the value
            }
        }
    }
}
```

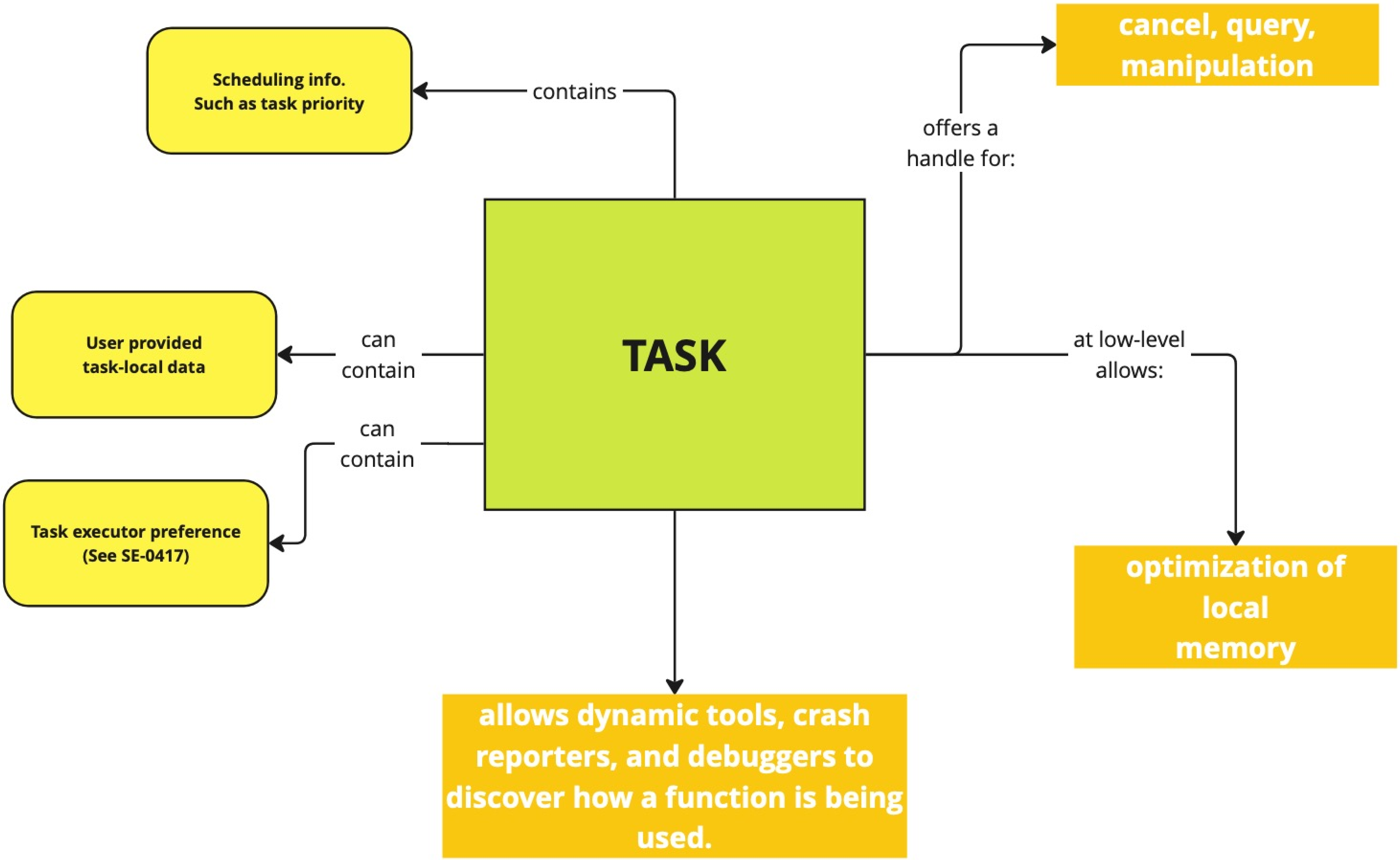
Structure concurrency, tasks, groups SE-0304 SE-0311 SE-0323 SE-0343 SE-0381 SE-417 SE-442



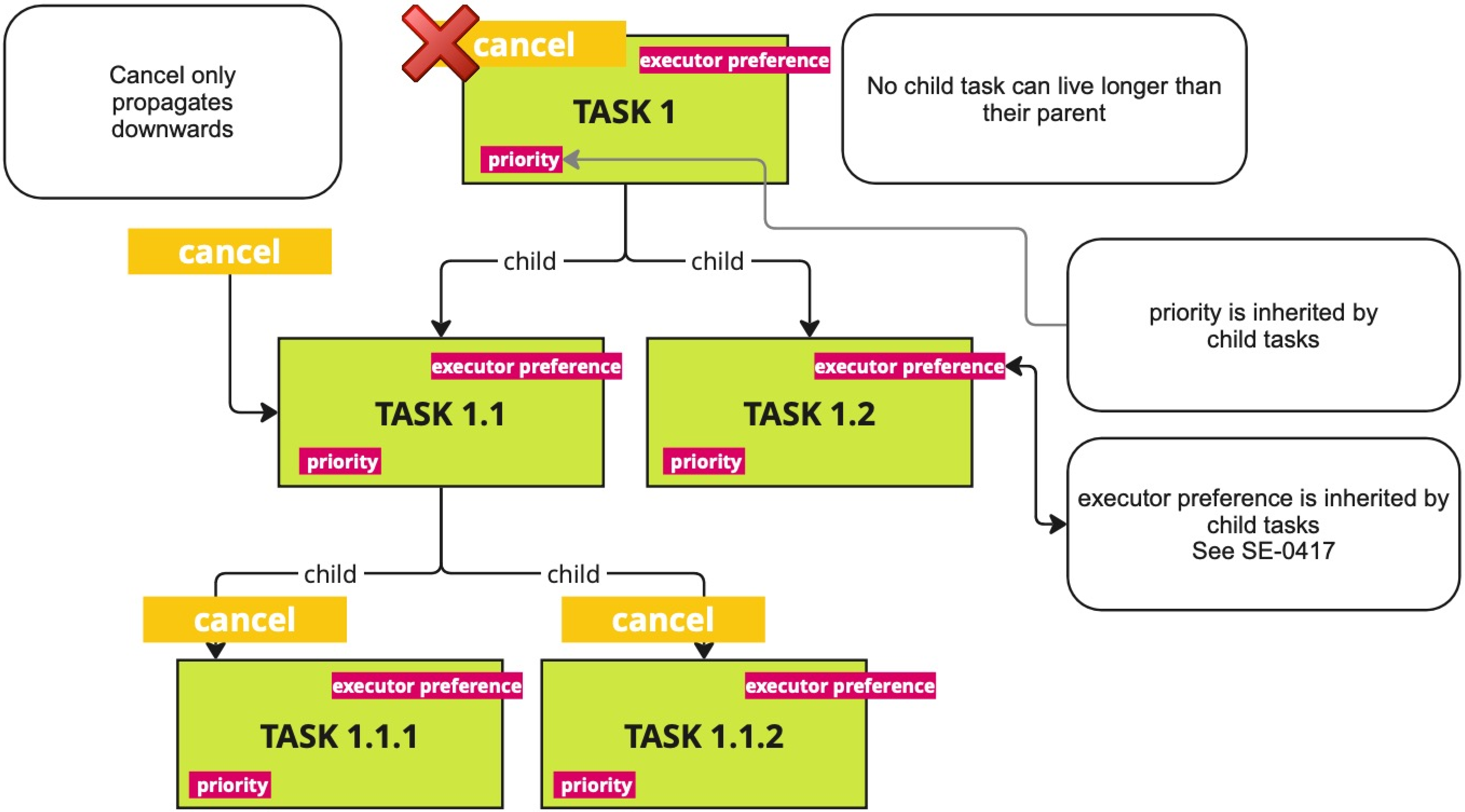
Task's States



Task's Structure

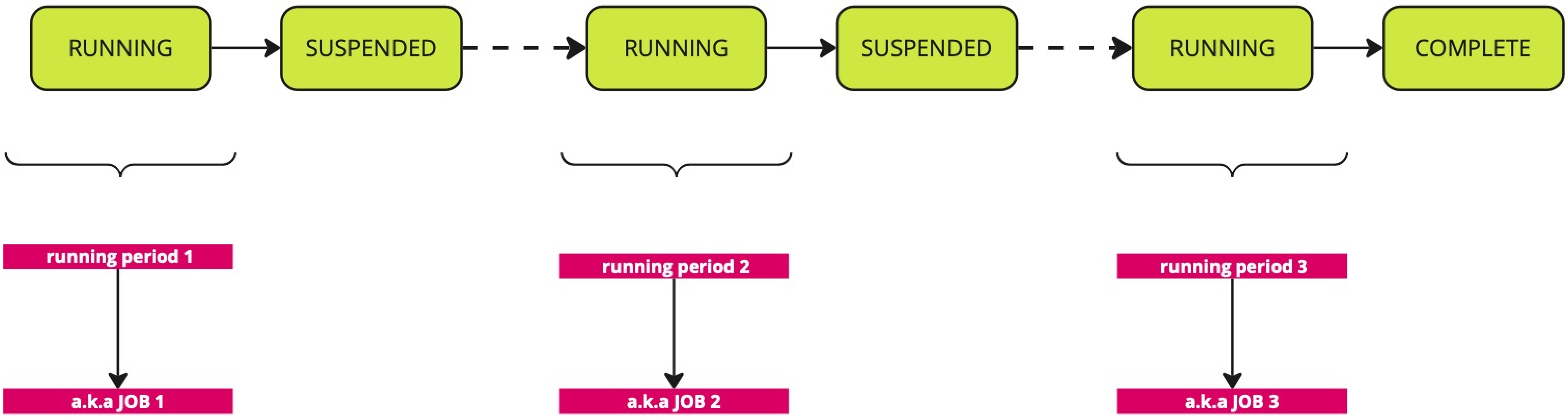


Taks and child tasks



Taks and jobs

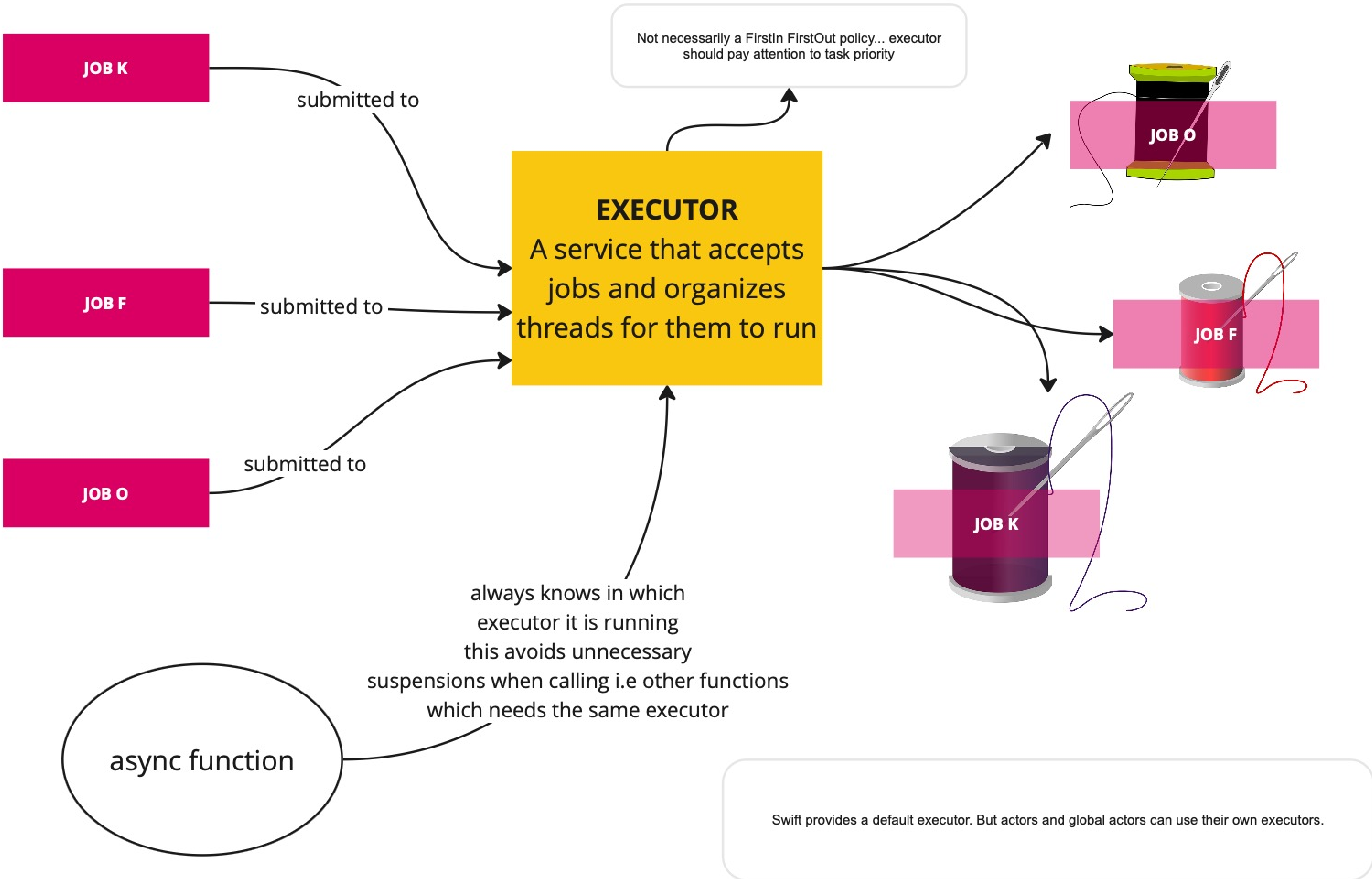
A task run in a succession of periods, till it is suspended, and then runs again.
Those running periods are called JOBS



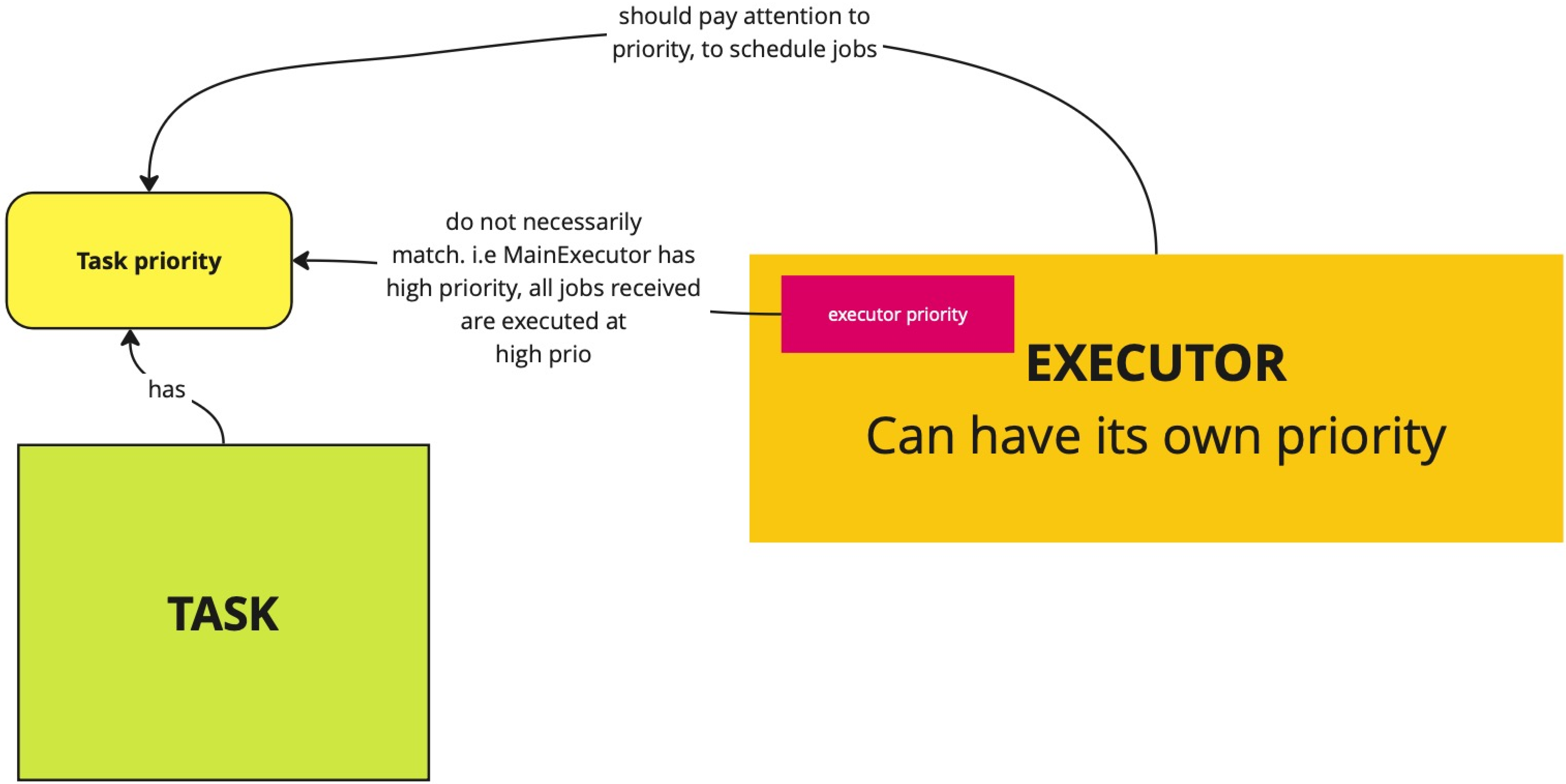
JOB = BASIC UNIT OF SCHEDULABLE WORK

YOU ONLY NEED TO BE AWARE OF JOBS WHEN YOU IMPLEMENT YOU OWN EXECUTOR

Executors


JOB K
JOB F
JOB O
submitted to
submitted to
submitted to
EXECUTOR
A service that accepts jobs and organizes threads for them to run
Not necessarily a FirstIn FirstOut policy... executor should pay attention to task priority
JOB O
JOB F
JOB K
async function
always knows in which executor it is running
this avoids unnecessary suspensions when calling i.e other functions which needs the same executor
Swift provides a default executor. But actors and global actors can use their own executors.

Executors and priority



Task and main

```swift
// Swift will create a new task that will execute main().
// Once that task completes, the program terminates.
@main
struct SwiftConcurrencyConceptualMapsApp: App {
    var body: some Scene {
        WindowGroup {
            ContentView()
        }
    }
}
```

Task and cancellation

CAN BE **EXPLICIT**

BY CALLING CANCEL on a task handler

CAN BE **AUTOMATIC**

when the parent task is cancelled

TASK

flag = cancelled

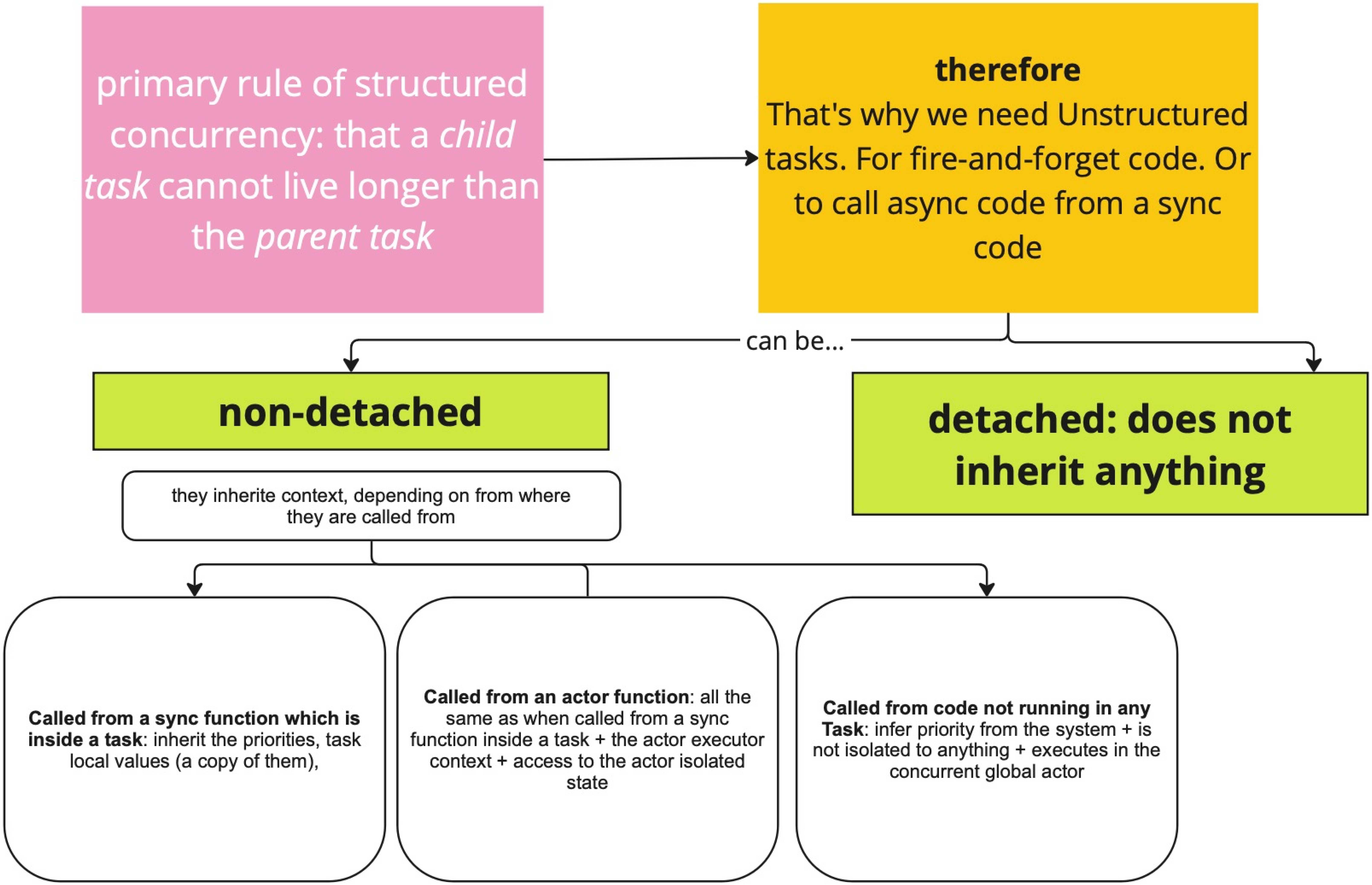
on cancellation a flag is set forever on the task, that says it is cancelled

```swift
struct Vegetable: Ingredient {
    func chopped() async throws -> Vegetable {
        try? await Task.sleep(for: .seconds(0.3))
        // if this task would be long, we need to check for cancellations
        try Task.checkCancellation()
        // if it is still not cancelled, we can continue
        return self
    }
}
```

```swift
loadingTask = Task {
    // some very long calculation here....
    // or if you need to react immediately to
    // the cancellation, create a
    // task with cancellation handler
    await withTaskCancellationHandler {
        // some long processing here
    } onCancel: {
        // do some cleanup
    }
}
```

if you register a cancellation handler, then it will be called immediately after cancellation, even if inside the code you do not check for cancellation

Unstructured task



About TaskGroup and DiscardingTaskGroup

Task groups are the building block of structured concurrency. They enable features such as automatic cancellation propagation, propagating errors, and ensuring well-defined lifetimes

but they have a limitation
since they are expected to return the results, they should keep them in memory until they are consumed, this can blow out the memory space if results are not consumed and a task group is potentially infinite (i.e. server attending requests loop)

some attempts to solve this, before the addition of DiscardingTaskGroups involved limiting the max capacity of a group... but manually

```
try await withThrowingTaskGroup(of: Void.self) { group in
    // Fill the task group up to maxConcurrency
    for _ in 0..<maxConcurrency {
            guard let newConnection = try await listeningSocket.accept() else {
                break
            }

            group.addTask { handleConnection(newConnection) }
        }

        // Now follow a one-in-one-out pattern
        while true {
            _ = try await group.next()
            guard let newConnection = try await listeningSocket.accept() else {
                break
            }
            group.addTask { handleConnection(newConnection) }
        }
}
```

Discarding groups, do this automatically... they clean up tasks when completed, BUT therefore can not have a next() function NOR have ChildTaskResult

```
// GOOD, no leaks!
try await withThrowingDiscardingTaskGroup() { group in
    while let newConnection = try await listeningSocket.accept() {
        group.addTask {
            handleConnection(newConnection)
        }
    }
}
```

Where do async functions run?

SE-0296

now we have **async functions**

SE-0306

now we have **actors and actor-isolated async functions may run in their actor's executor** and also non-isolated functions won't change executor, so they may also use the actor's executor

SE-0338

but then **non-isolated async functions** may over-populate the actor's executor... so now let's make them FORMALLY **run in a generic, non-actor global executor**

HOP-OFF SEMANTIC INTRODUCTION

formally = in theory... in reality internally the system can decide if to do the switch or not for i.e optimization

SE-0417

Swift 6.0

but now, there will be lots of HOP-OFFS potentially...so if you need to not switch context so often... **TASK EXECUTOR PREFERENCE is introduced and you can choose**

SE-0420

Swift 6.0

but these hop-offs can be very inefficient... so let's give developers the chance to do something to indicate that the actor context has to be inherited also for non-isolated functions....**ACTOR EXECUTOR INHERITANCE**

SE-0417 Task executor preference

Swift 6.0

DO NOT USE TASK EXECUTOR PREFERENCE UNLESS YOU REALLY NEED IT FOR PERFORMANCE

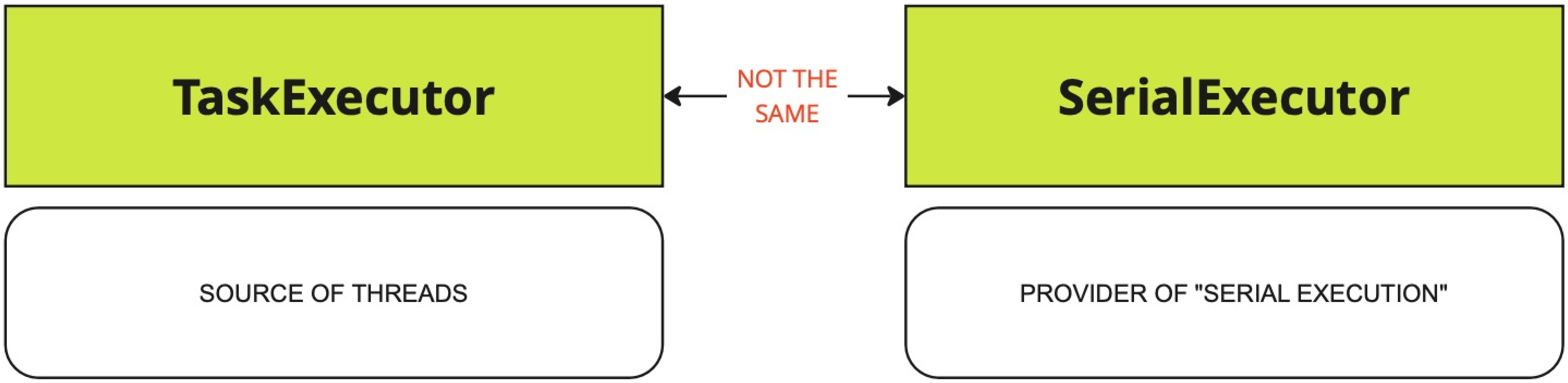
SE-0338

said that non-actor isolated function will run on the global generic executor. But in case your codebase has a real need to extreme performance, SE-0417 allows you to avoid the hops out of an actor

```
// `func` execution semantics with this proposal

[ func / closure ] - /* where should it execute? */
                               |
                     +--------------+          +==========================+
           +-------- | is isolated? | - yes -> | actor has unownedExecutor |
           |         +--------------+          +==========================+
           |                                        |                |
           |                                       yes               no
           |                                        |                |
           |                                        v                v
           |                  +=======================+    /* task executor preference? */
           |                  | on specified executor |        |                   |
           |                  +=======================+       yes                  no
           |                                                   |                   |
           |                                                   |                   v
           |                                                   |    +==========================+
           |                                                   |    | default (actor) executor |
           |                                                   v    +==========================+
           v                                   +==============================+
/* task executor preference? */ ---- yes ----> | on Task's preferred executor |
           |                                   +==============================+
           no
           |
           v
  +===============================+
  | on global concurrent executor |
  +===============================+
```

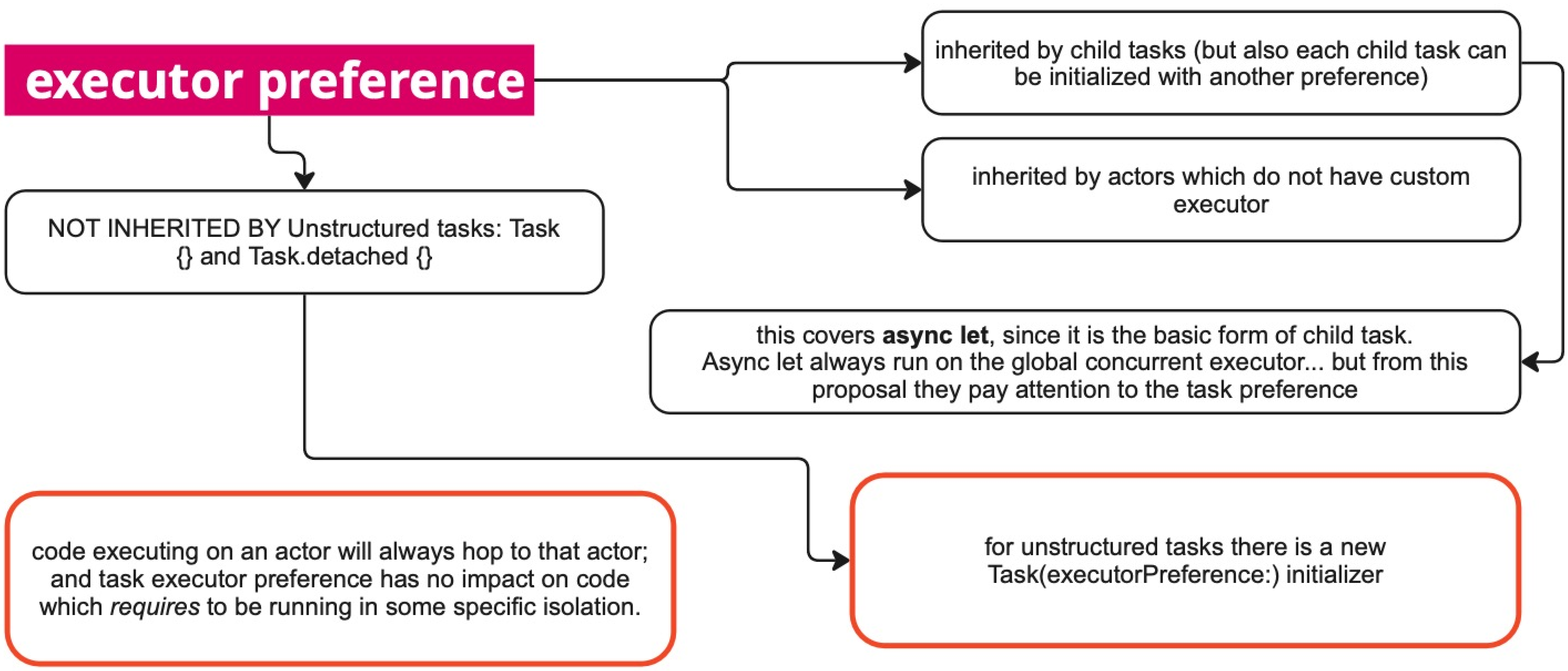
there is a new protocol

```swift
public protocol TaskExecutor: Executor {
  func enqueue(_ job: consuming ExecutorJob)

  func asUnownedTaskExecutor() -> UnownedTaskExecutor
}
```




```swift
actor RunsAnywhere { // a "default" actor == without an executor requirement
  func hello() {
    return "Hello"
  }
}
```

```swift
let anywhere = RunsAnywhere()
Task { await anywhere.hello() } // runs on "default executor", using a thread from the global pool

Task(executorPreference: myExecutor) {
  // runs on preferred executor, using a thread owned by that executor
  await anywhere.hello()
}
```




```swift
// the global concurrent executor can be accessed like this
// and can be used a preference, to 'deactivate' any ihnerited
// previously set preference
print(globalConcurrentExecutor)
```

SE-0420 Inheritance of Actor Isolation

Swift 6.0

SE-0338 defined that non isolated async functions in acts will hop to the global concurrent actor. This can be inefficient and this proposal adds support for developers to explicitly choose to inherit the actor isolation

actor-isolated properties

'isolated' allows context inheritance to happen

here for example without 'isolated' the compilation fails

'isolated' make the debug function isolated to the actor
So now, we must call it with 'await'

```swift
// An actor
actor Course {
    // with 2 actor-isolated propertie
    var name = "Maths"
    var studentNames = [String]()
}

// another function with a parameter
// marked as 'isolated'
func debugLogWithActorContextInheritance(course: isolated Course) {
    // it is possible to access actor-
    // isolated parameters
    print("Name: \(course.name)")
    print("Students: \(course.studentNames)")
}

// another function with no parameter
// marked as 'isolated'
func debugLog(course: Course) {
    // it is possible to access actor-
    // isolated parameters
    print("Name: \(course.name)")
    print("Students: \(course.studentNames)")
}

func exampleContextInheritance() {
    Task {
        let course = Course()
        // but now we need to use await even
        // if debugLog is not async
        debugLogWithActorContextInheritance(course: course)
    }
}
```

Actor-isolated property 'name' can not be referenced from a...

Actor-isolated property 'studentNames' can...

Expression is 'async' but i...