The Concurrency Runtime

Braden Scothern - SLC CocoaHeads - April 4, 2023

What is concurrency?

What is concurrency?

Having multiple tasks making progress at the same time

What is concurrency?

Having multiple tasks making progress at the same time

What is parallelism?

What is concurrency?

Having multiple tasks making progress at the same time

What is parallelism?

Having multiple threads executing at the same time

Swift’s Goals

1. Convenient

https://forums.swift.org/t/swift-concurrency-roadmap

Swift’s Goals

1. Convenient
2. Efficient

https://forums.swift.org/t/swift-concurrency-roadmap

Swift’s Goals

1. Convenient
2. Efficient
3. Safe

https://forums.swift.org/t/swift-concurrency-roadmap

Swift's Goals

and what it means for the runtime

1. ~~Convenient~~ Details are hidden as much as possible
2. Efficient
3. Safe

https://forums.swift.org/t/swift-concurrency-roadmap

Swift's Goals

and what it means for the runtime

1. ~~Convenient~~ Details are hidden as much as possible
2. ~~Efficient~~ Complexity with minimal overhead behind the scenes
3. Safe

https://forums.swift.org/t/swift-concurrency-roadmap

Swift's Goals

and what it means for the runtime

1. ~~Convenient~~ Details are hidden as much as possible
2. ~~Efficient~~ Complexity with minimal overhead behind the scenes
3. ~~Safe~~ Static analysis during compilation and/or overhead

https://forums.swift.org/t/swift-concurrency-roadmap

Coroutines

Coroutine Implementation Challenges

- Control Changes
- Storing State
- Yielding Values

Control Changes
Context Switching

- Single Function
- Yield saves registers
- Resume restores registers
- Lower Code Size
- Ties up a stack
- Higher time/space overhead

Control Changes
Coroutine splitting

- Split coroutine into many sub-functions
- Lives on normal stack
- Yield returns from current sub-function
- Resume calls the next sub-function
- “Ramp” sub-function takes original parameters
- 2 Main Styles with their own trade-offs
  - Double-dispatch or potentially quadratic code-size

Storing State
Stackful

- A full stack is created during the “ramp” function
- Deallocated when done
- Context-switch to get state
- No splitting which saves size
- All the disadvantages of a full callstack

Storing State
Side Allocation

- Requires splitting
- Separate allocations for cross-yield data
- Uses lots of heap allocation

Storing State
Stack cohabitation

- “Normal” function call
  - It doesn’t shrink the stack on yield like a return would
- Very complex code generation
- Odd when you have multiple coroutines
- Tied to a single stack
  - Can potentially be promoted to a stackful state coroutine

Yielding Values

Direct Return

- Good when used immediately (generators, for await loops)

Fixed Location

- Good when used later (async/await)

Swift's Approach

- Specialize Generators
- Rapid back-and-forth with caller
- Prioritize efficient access to yield value
- May cross async yields

Where coroutines show up in are code
(At least the most common places)

- Async/Await
- _modify accessors on properties
  - This is often synthesized by the compiler for simple cases of get+set
- for await
  - These are a kind of generator from the previous slide

Executors

SE-0392: Custom Actor Executors

Returned for revision

- Where this interface is being finalized and made a bit nicer
- Everything in the following slides is already in Swift

Executor protocol

- How you control where async work happens

- Each piece of work is currently called a Job
  - This is changing as part of the current SE-0392’s revisions
  - These are changing to move-only types

Protocol

Executor

A service that can execute jobs.

iOS 13.0+ | iPadOS 13.0+ | macOS 10.15+ | Mac Catalyst 13.0+ | tvOS 13.0+ | watchOS 6.0+

Declaration

```
protocol Executor : AnyObject, Sendable
```

Topics

Instance Methods

`func enqueue(UnownedJob)`

Required.

Relationships

Inherits From

`Sendable`

Inherited By

`SerialExecutor`