А так ли нужны акторы в Swift Concurrency?

Василий Усов

iOS-разработчик в Райффайзен банке
Автор книг по разработке на Swift
Увлекаюсь программированием более 20 лет

Райффайзен

VK Карты

А так ли нужна Swift Modern Concurrency

Swift Concurrency

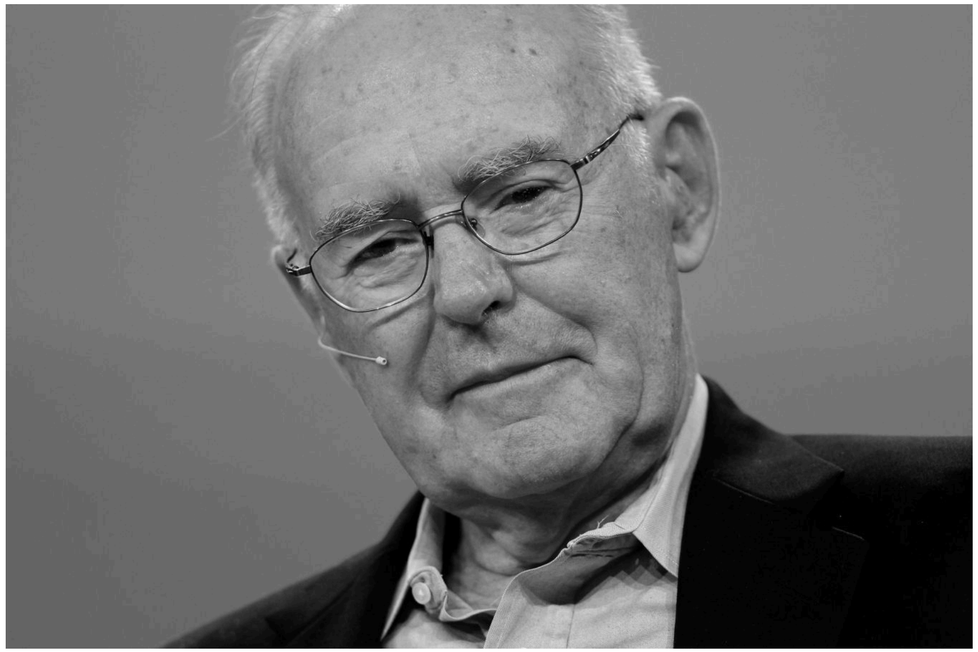
Task
Корутины

Async/await
F# и C#

Акторы

Акторы

Ретроспектива



Гордон Мур

Соучредитель Intel и
Fairchild Semiconductors,
автор Закона Мура

1965 Закон Мура

1960+ Появление концепций конкурентности, в том числе многопоточности и корутин


Изменяемое
состояние

1965	Закон Мура
1960+	Появление концепций конкурентности, в том числе многопоточности и корутин
1983	Objective-C
1996	Стандарт POSIX Threads Extension (IEEE Std 1003.1c-1995), C-библиотека pthread


Изменяемое
состояние

Изменяемое состояние

```
var state = 0
var mutex = pthread_mutex_t()
pthread_mutex_init(&mutex, nil)

func mutateState(_ closure: (Int)->Int) {
    pthread_mutex_lock(&mutex);
    defer {
        pthread_mutex_unlock(&mutex);
    }
    state = closure(state)
}
```

```
var state = 0
var lock = NSLock()

func mutateState(_ closure: (Int)->Int) {
    lock.lock()
    defer {
        lock.unlock()
    }
    state = closure(state)
}
```

1965	Закон Мура
1960+	Появление концепций конкурентности, в том числе многопоточности и корутин
1983	Objective-C
1996	Стандарт POSIX Threads Extension (IEEE Std 1003.1c-1995), C-библиотека pthread
2003	Статья Мура «No Exponential is Forever: But „Forever“ Can Be Delayed!»
2008	OS X 10.6 Snow Leopard, C-библиотека libdispatch (Grand Central Dispatch)

Изменяемое состояние

```
var state = 0
var mutex = pthread_mutex_t()
pthread_mutex_init(&mutex, nil)

func mutateState(_ closure: (Int)->Int) {
    pthread_mutex_lock(&mutex);
    defer {
        pthread_mutex_unlock(&mutex);
    }
    state = closure(state)
}
```

```
var state = 0
var lock = NSLock()

func mutateState(_ closure: (Int)->Int) {
    lock.lock()
    defer {
        lock.unlock()
    }
    state = closure(state)
}
```

```
var state = 0
let serialQueue = DispatchQueue(label: "mutateQueue")

func mutateState(_ closure: @escaping (Int)->Int) {
    serialQueue.async {
        state = closure(state)
    }
}
```

Spinning Beachball of Death

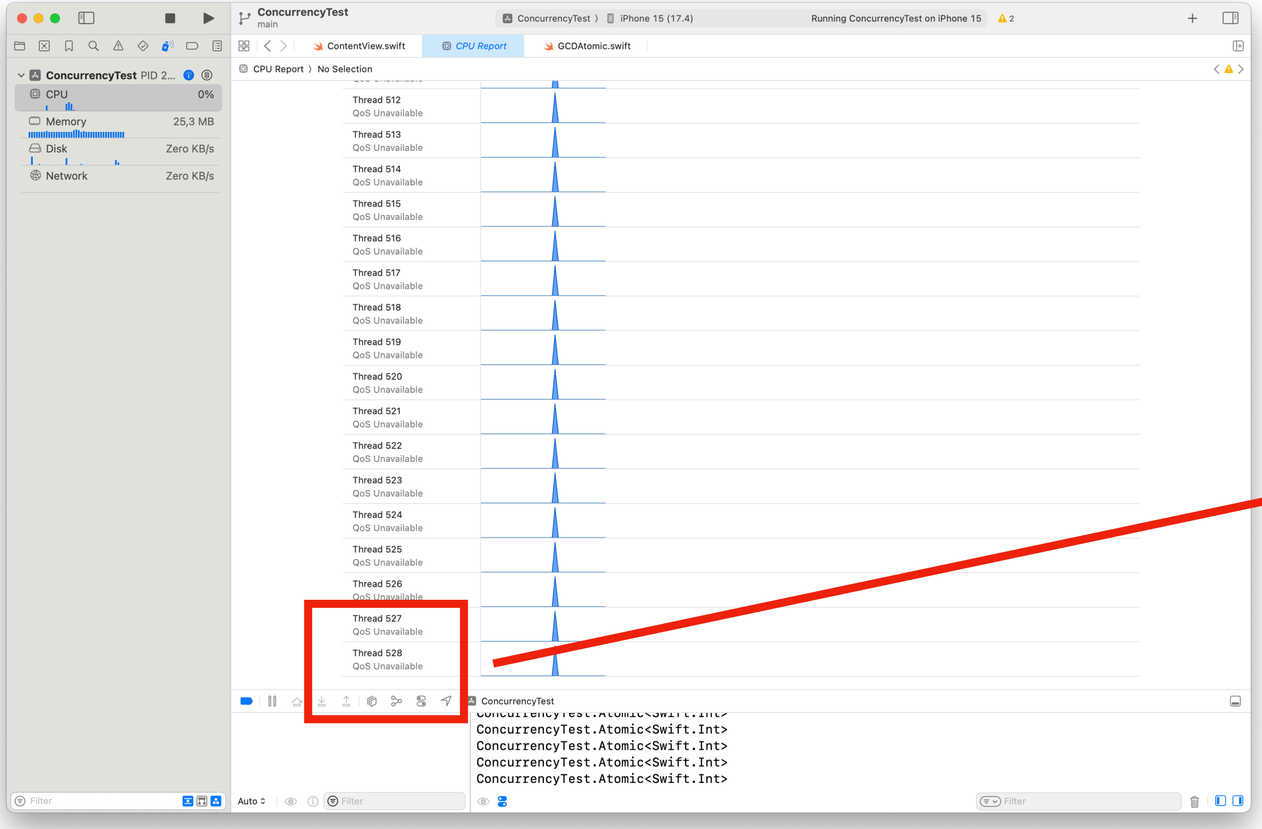
GCD Atomic

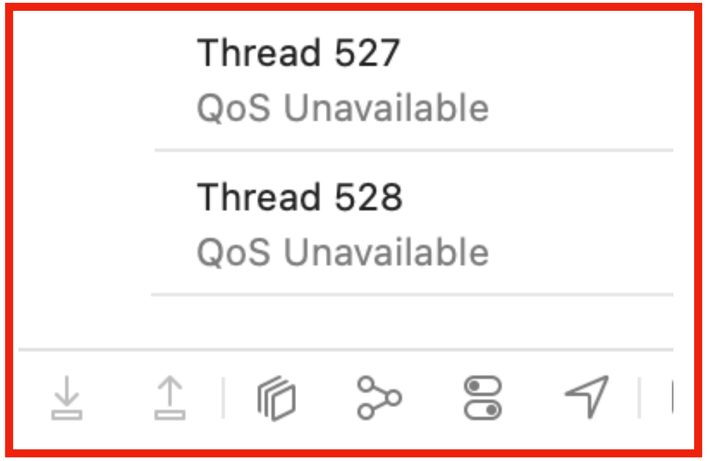
```
final class Atomic<Wrapped> {
    private let queue = DispatchQueue(label: "AtomicQueue")
    private var _value: Wrapped
    var value: Wrapped {
        queue.sync { _value }
    }

    init(value: Wrapped) {
        self._value = value:chQueue(label: "mutateQueue")
    }
    func mutateState(_ closure: @escaping (Int)->Int) {
    func mutate(_ handler: @escaping (Wrapped) -> Wrapped) {
        queue.async { [self] in:e)
            _value = handler(_value)
        }
    }

    func doSomething() {
        queue.async {
            // тут может изменяться _value
        }
    }
}
```

Атомарность

Синхронизация

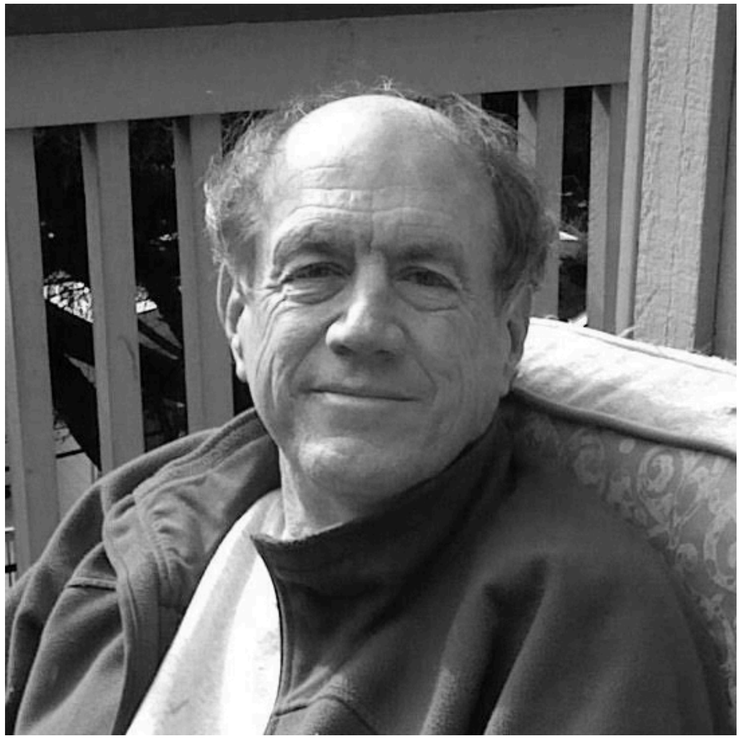
Ретроспектива





1965	Закон Мура
1960+	Появление концепций конкурентности, в том числе многопоточности и корутин
1983	Objective-C
1996	Стандарт POSIX Threads Extension (IEEE Std 1003.1c-1995), C-библиотека pthread
2003	Статья Мура «No Exponential is Forever: But „Forever“ Can Be Delayed!»
2008	OS X 10.6 Snow Leopard, C-библиотека libdispatch (Grand Central Dispatch)
2011	OS X 10.7 Lion, Библиотека Security Transform
2014	Swift
2017	Swift Concurrency Manifesto
2021	Swift Concurrency в составе Swift 5.5

Акторы



Карл Хьюитт

Учетный в области информатики, автор языка программирования Planner и модели акторов

«Planner: A language for proving theorems in robots», 1969

Описание языка автоматического планирования и диспетчеризации в робототехнике

«A Universal Modular ACTOR Formalism for Artificial Intelligence», 1973

Описание концепции акторов, используемой в проекте Planner

«Actor Induction and Meta-evaluation», 1974

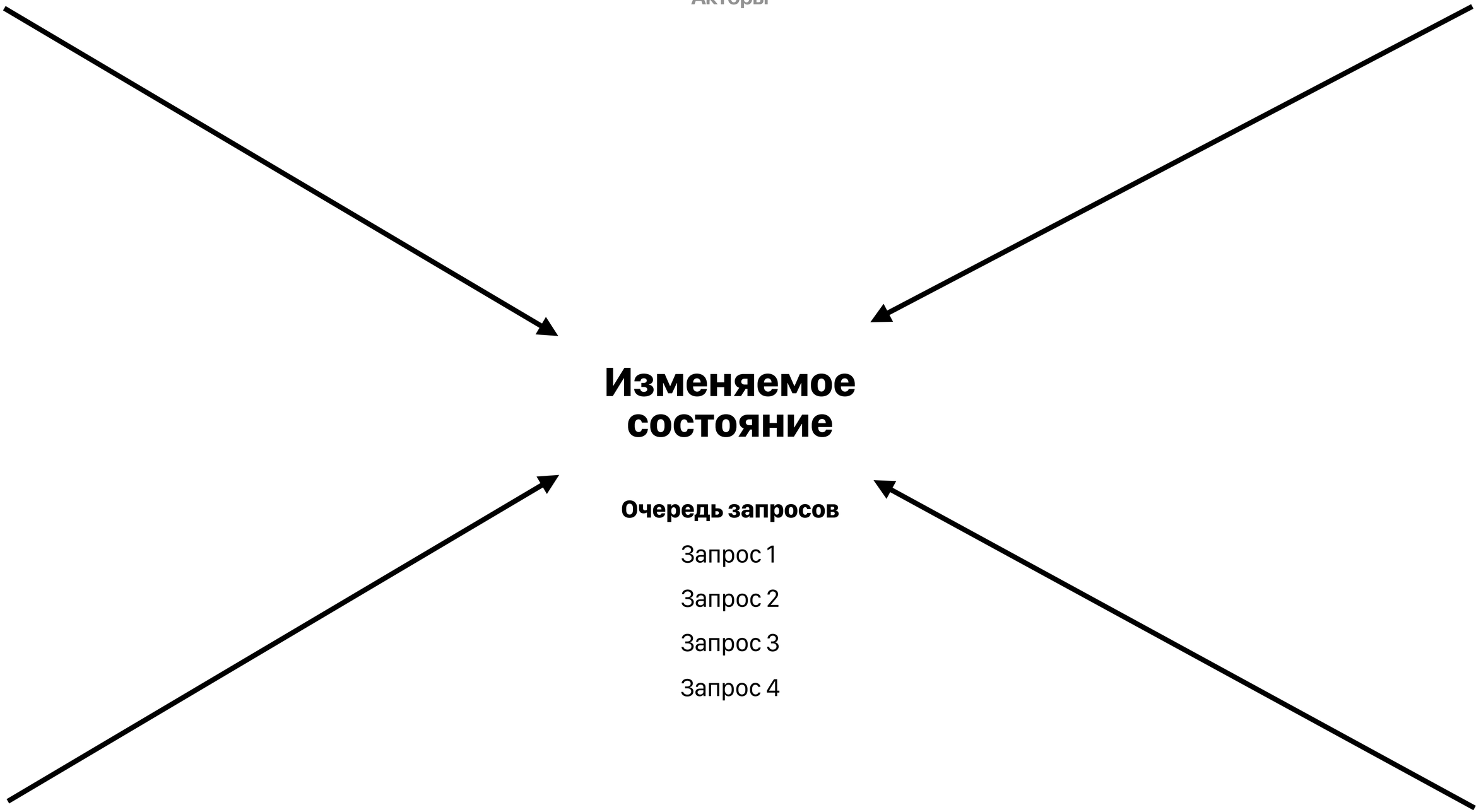
«Specification and Verification Techniques for Parallel Programs Based on Message Passing Semantics», 1977

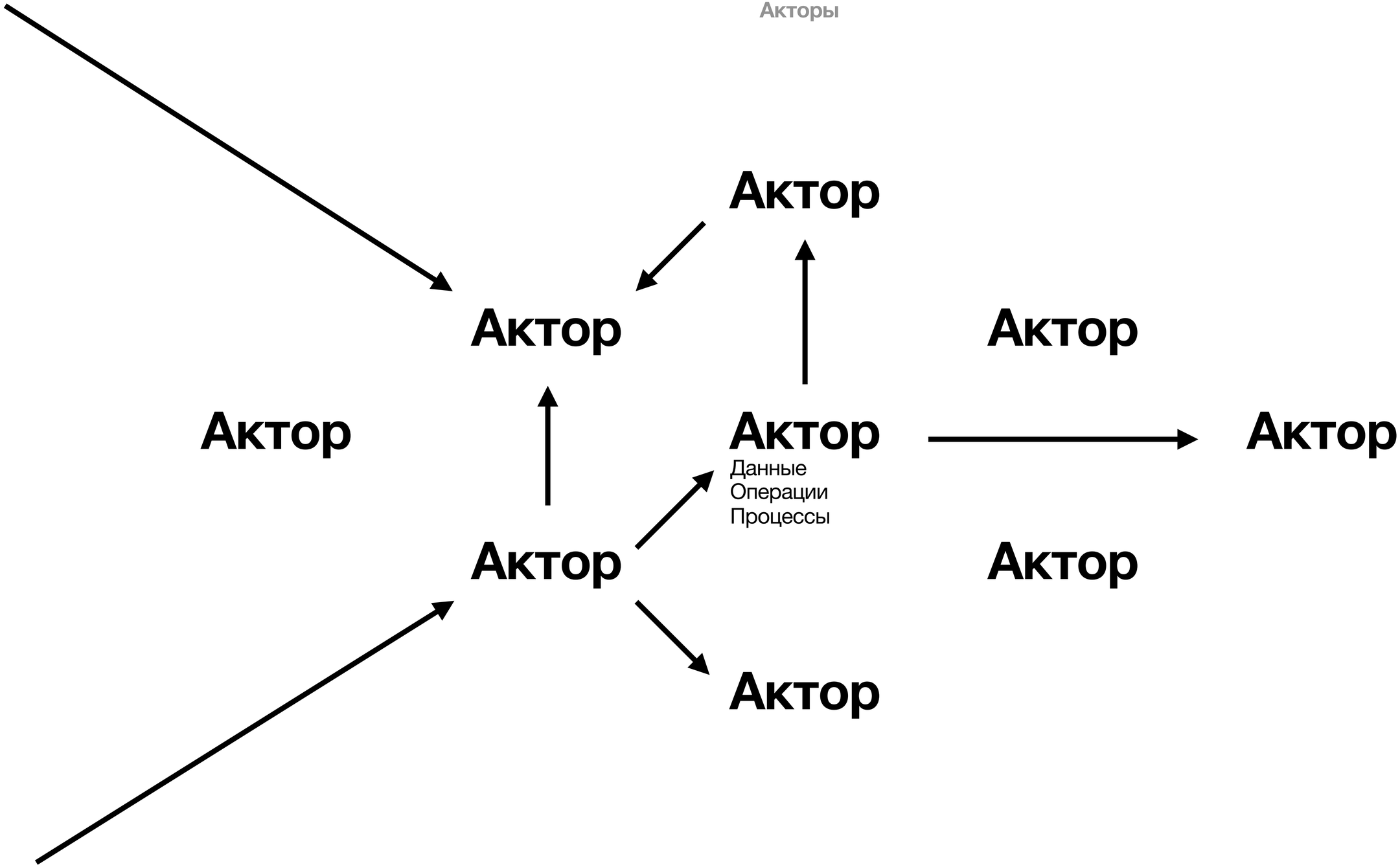
«Actor Systems for Real-Time Computation», 1978

«Thinking About Lots of Things at Once without Getting Confused: Parallelism in Act 1», 1981

«Foundations of Actor Semantics», 1981

«From objects to actors: Study of a limited symbiosis in Smalltalk-80», 1988


Изменяемое состояние
Очередь запросов
Запрос 1
Запрос 2
Запрос 3
Запрос 4


Актор
Актор
Актор
Актор
Актор
Актор
Данные
Операции
Процессы
Актор
Актор
Актор

Swift Concurrency Atomic

```swift
actor AtomicActor<Wrapped> {
    var value: Wrapped

    init(value: Wrapped) {
        self.value = value
    }

    func mutate(_ handler: (Wrapped) -> Wrapped) {
        value = handler(value)
    }

    func doSomething() {
        // тут читается и изменяется Стейт
    }
}

// Использование
let actor = AtomicActor(value: 2)
Task {
    await actor.mutate { value in
        value + 1
    }
    await actor.value
    await actor.doSomething()
}
```

Изоляция

Swift Concurrency Atomic

```
actor AtomicActor<Wrapped> {
    var value: Wrapped

    init(value: Wrapped) {
        self.value = value
    }

    func mutate(_ handler: (Wrapped) -> Wrapped) {
        value = handler(value)
    }

    func doSomething() {
        // тут читается и изменяется Стейт
    }

    nonisolated func doSomethingOther() {
        // этот метод не изолирован на акторе
    }
}

// Использование
let actor = AtomicActor(value: 2)
Task {
    await actor.mutate { value in
        value + 1
    }
    await actor.value
    await actor.doSomething()
}
actor.doSomethingOther()
```

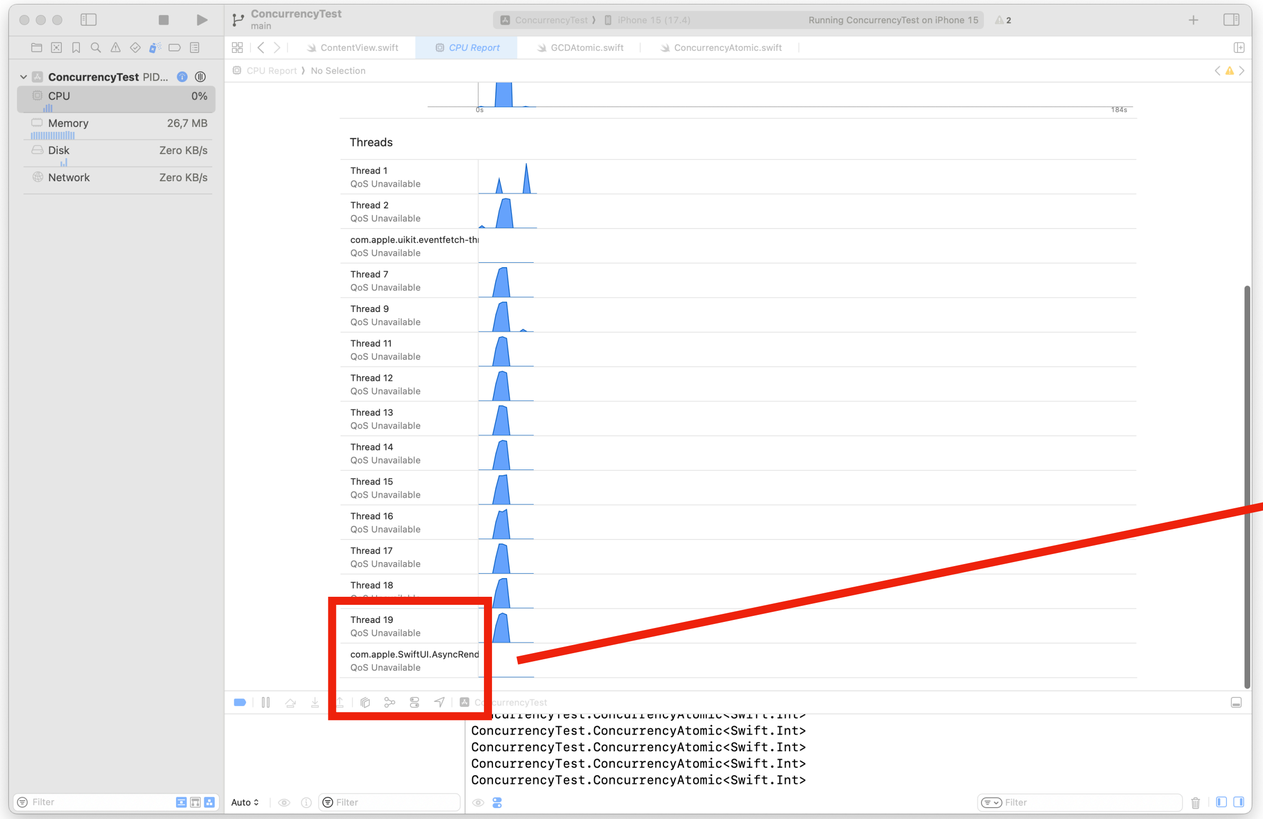
Данные

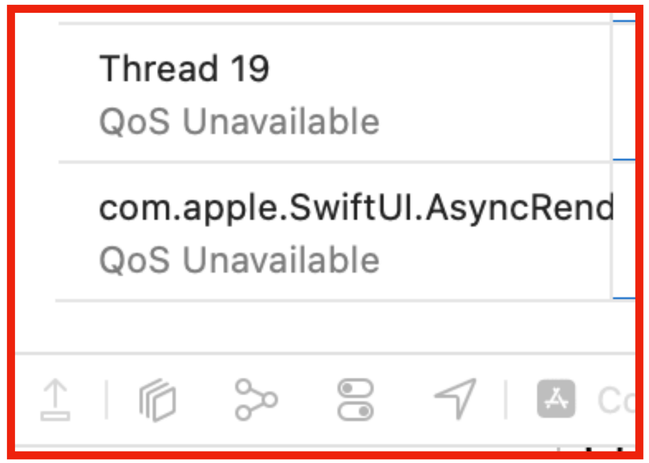
Операции

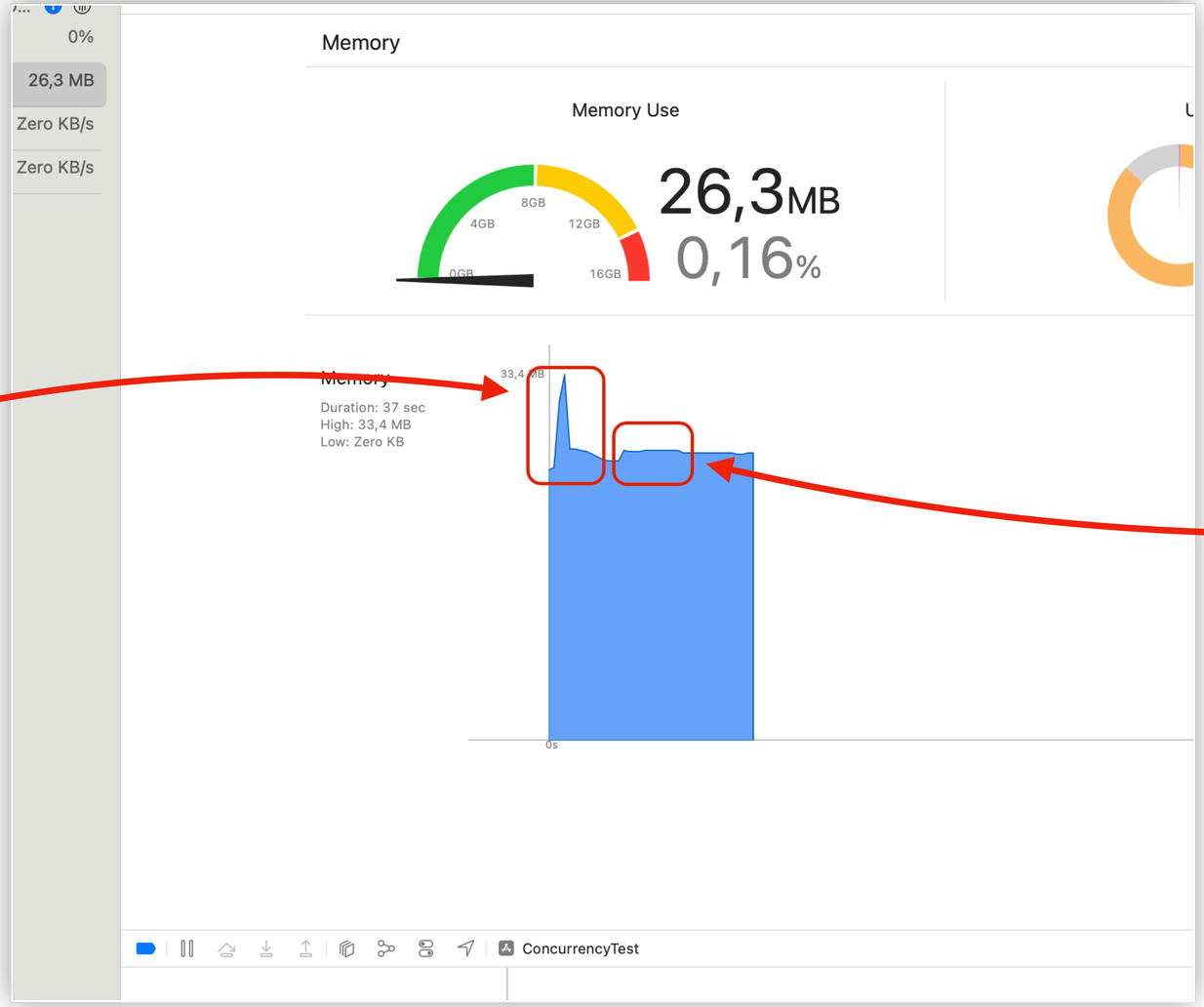
Процессы

Изоляция

Акторы


ConcurrencyTest
main
ConcurrencyTest PID...
CPU 0%
Memory 26,7 MB
Disk Zero KB/s
Network Zero KB/s
ContentView.swift
CPU Report
GCDAtomic.swift
ConcurrencyAtomic.swift
CPU Report No Selection
Threads
Thread 1
QoS Unavailable
Thread 2
QoS Unavailable
com.apple.uikit.eventfetch-th
QoS Unavailable
Thread 7
QoS Unavailable
Thread 9
QoS Unavailable
Thread 11
QoS Unavailable
Thread 12
QoS Unavailable
Thread 13
QoS Unavailable
Thread 14
QoS Unavailable
Thread 15
QoS Unavailable
Thread 16
QoS Unavailable
Thread 17
QoS Unavailable
Thread 18
Thread 19
QoS Unavailable
com.apple.SwiftUI.AsyncRend
QoS Unavailable
ConcurrencyTest.ConcurrencyAtomic<Swift.Int>
ConcurrencyTest.ConcurrencyAtomic<Swift.Int>
ConcurrencyTest.ConcurrencyAtomic<Swift.Int>
ConcurrencyTest.ConcurrencyAtomic<Swift.Int>
Auto
Filter


Thread 19
QoS Unavailable
com.apple.SwiftUI.AsyncRend
QoS Unavailable


0%
26,3 MB
Zero KB/s
Zero KB/s
Memory
Memory Use
26,3MB
0,16%
0GB
4GB
8GB
12GB
16GB
Memory
Duration: 37 sec
High: 33,4 MB
Low: Zero KB
33,4 MB
0s
ConcurrencyTest

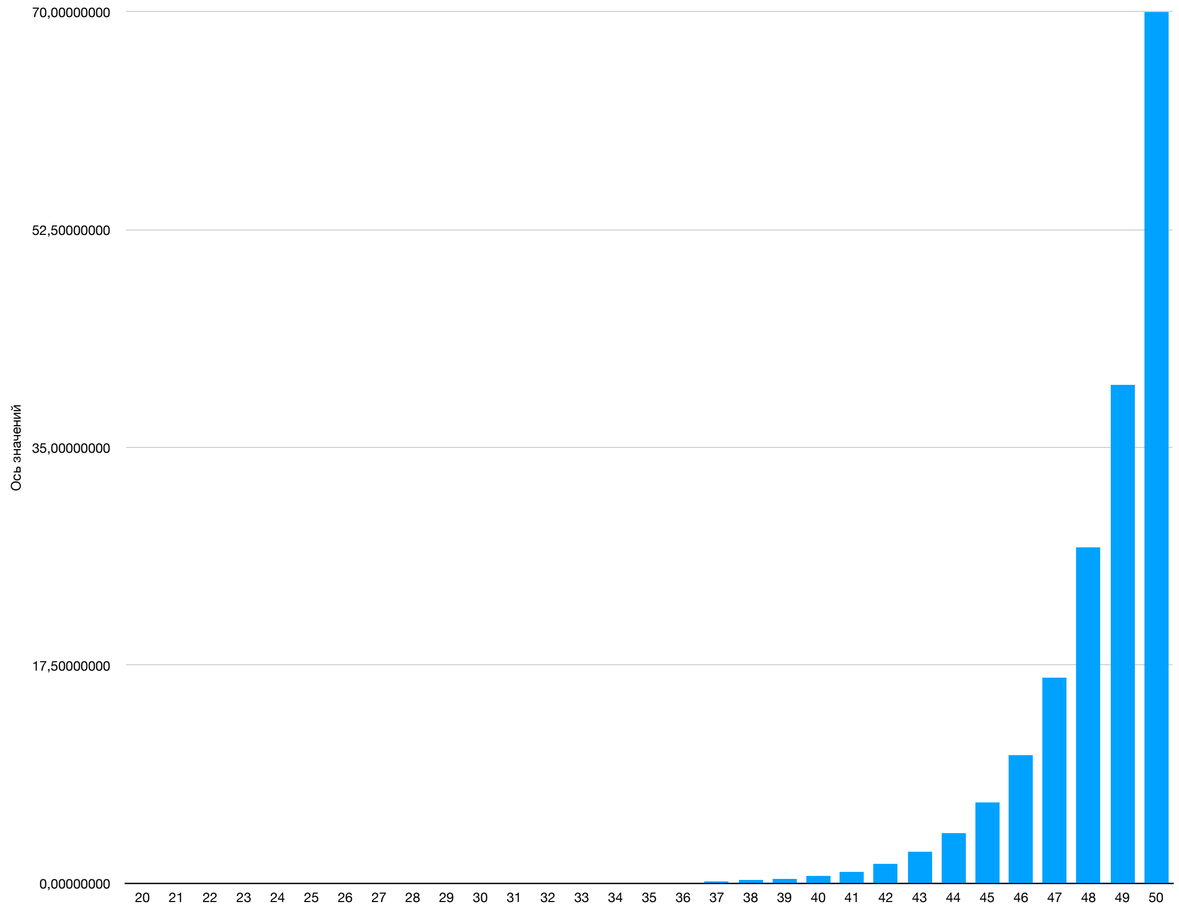
GCD

Actor

Особенности использования

Особенности использования

```
for i in 20...50 {
    Task {
        _ = await calculateFibbonachi(i)
    }
}


func calculateFibbonachi(_ number: Int) async -> Int {
    switch number {
    case 1: return 0
    case 2: return 1
    default:
        let num1 = await calculateFibbonachi(number - 1)
        let num2 = await calculateFibbonachi(number - 2)
        await Task.yield()
        return num1 + num2
    }
}
```



Отсутствие инверсии приоритетов

Отсутствие инверсии приоритетов

```swift
let actor = AtomicActor(value: 0)
for i in 1...30 {
    let randomPriority = [TaskPriority.low, .high].randomElement()!
    Task(priority: randomPriority) {
        await actor.mutate {
            print("PRIORITY: \(Task.currentPriority), VALUE: \(i)")
            return $0 + i
        }
    }
}
```

Консоль

```
PRIORITY: TaskPriority.low, VALUE: 1
PRIORITY: TaskPriority.high, VALUE: 2
PRIORITY: TaskPriority.high, VALUE: 5
PRIORITY: TaskPriority.high, VALUE: 6
PRIORITY: TaskPriority.high, VALUE: 7
PRIORITY: TaskPriority.high, VALUE: 10
PRIORITY: TaskPriority.high, VALUE: 14
PRIORITY: TaskPriority.high, VALUE: 15
PRIORITY: TaskPriority.high, VALUE: 17
PRIORITY: TaskPriority.high, VALUE: 18
PRIORITY: TaskPriority.high, VALUE: 20
PRIORITY: TaskPriority.high, VALUE: 24
PRIORITY: TaskPriority.high, VALUE: 25
PRIORITY: TaskPriority.high, VALUE: 26
PRIORITY: TaskPriority.high, VALUE: 30
PRIORITY: TaskPriority.low, VALUE: 3
PRIORITY: TaskPriority.low, VALUE: 4
PRIORITY: TaskPriority.low, VALUE: 8
PRIORITY: TaskPriority.low, VALUE: 9
PRIORITY: TaskPriority.low, VALUE: 11
PRIORITY: TaskPriority.low, VALUE: 12
PRIORITY: TaskPriority.low, VALUE: 13
PRIORITY: TaskPriority.low, VALUE: 16
PRIORITY: TaskPriority.low, VALUE: 19
PRIORITY: TaskPriority.low, VALUE: 21
PRIORITY: TaskPriority.low, VALUE: 22
PRIORITY: TaskPriority.low, VALUE: 23
PRIORITY: TaskPriority.low, VALUE: 27
PRIORITY: TaskPriority.low, VALUE: 28
PRIORITY: TaskPriority.low, VALUE: 29
```

Проблемы атомарности

Проблемы атомарности

```
actor NetworkService {
    var currentTask: URLSessionDataTask? = nil

    // сделать запрос
    func make(request: URLRequest) async -> Data? {
        let result = await withCheckedContinuation { continuation in
            currentTask = URLSession.shared.dataTask(with: request,
                                                     completionHandler: { data, _, _ in
                continuation.resume(returning: data)
            })
        }
        currentTask = nil
        return result
    }

    // отменить запрос
    func cancelRequest() {
        currentTask?.cancel()
        currentTask = nil
    }
}
```

currentTask = `nil`

Исполняется	**Очередь запросов**
	make(request:)

Проблемы атомарности

```
actor NetworkService {
    var currentTask: URLSessionDataTask? = nil

    // сделать запрос
    func make(request: URLRequest) async -> Data? {
        let result = await withCheckedContinuation { continuation in
            currentTask = URLSession.shared.dataTask(with: request,
                                                     completionHandler: { data, _, _ in
                continuation.resume(returning: data)
            })
        }
        currentTask = nil
        return result
    }

    // отменить запрос
    func cancelRequest() {
        currentTask?.cancel()
        currentTask = nil
    }
}
```

currentTask = 1

Исполняется	Очередь запросов
1 - make(request:)	

Проблемы атомарности

```
actor NetworkService {
    var currentTask: URLSessionDataTask? = nil

    // сделать запрос
    func make(request: URLRequest) async -> Data? {
        let result = await withCheckedContinuation { continuation in
            currentTask = URLSession.shared.dataTask(with: request,
                                                     completionHandler: { data, _, _ in
                continuation.resume(returning: data)
            })
        }
        currentTask = nil
        return result
    }

    // отменить запрос
    func cancelRequest() {
        currentTask?.cancel()
        currentTask = nil
    }
}
```

currentTask = `nil`

Исполняется **Очередь запросов**

Проблемы атомарности

```
actor NetworkService {
    var currentTask: URLSessionDataTask? = nil

    // сделать запрос
    func make(request: URLRequest) async -> Data? {
        let result = await withCheckedContinuation { continuation in
            currentTask = URLSession.shared.dataTask(with: request,
                                                     completionHandler: { data, _, _ in
                continuation.resume(returning: data)
            })
        }
        currentTask = nil
        return result
    }

    // отменить запрос
    func cancelRequest() {
        currentTask?.cancel()
        currentTask = nil
    }
}
```

currentTask = `nil`

Исполняется	Очередь запросов
	make(request:)
	make(request:)

Проблемы атомарности

```
actor NetworkService {
    var currentTask: URLSessionDataTask? = nil

    // сделать запрос
    func make(request: URLRequest) async -> Data? {
        let result = await withCheckedContinuation { continuation in
            currentTask = URLSession.shared.dataTask(with: request,
                                                     completionHandler: { data, _, _ in
                continuation.resume(returning: data)
            })
        }
        currentTask = nil
        return result
    }

    // отменить запрос
    func cancelRequest() {
        currentTask?.cancel()
        currentTask = nil
    }
}
```

currentTask = 2

Исполняется	Очередь запросов
make(request:)	
	make(request:)

Проблемы атомарности

```
actor NetworkService {
    var currentTask: URLSessionDataTask? = nil

    // сделать запрос
    func make(request: URLRequest) async -> Data? {
        let result = await withCheckedContinuation { continuation in
            currentTask = URLSession.shared.dataTask(with: request,
                                                     completionHandler: { data, _, _ in
                continuation.resume(returning: data)
            })
        }
        currentTask = nil
        return result
    }

    // отменить запрос
    func cancelRequest() {
        currentTask?.cancel()
        currentTask = nil
    }
}
```

currentTask = 2

Исполняется	Очередь запросов
make(request:)	
	make(request:)

Проблемы атомарности

```
actor NetworkService {
    var currentTask: URLSessionDataTask? = nil

    // сделать запрос
    func make(request: URLRequest) async -> Data? {
        let result = await withCheckedContinuation { continuation in
            currentTask = URLSession.shared.dataTask(with: request,
                                                     completionHandler: { data, _, _ in
                continuation.resume(returning: data)
            })
        }
        currentTask = nil
        return result
    }

    // отменить запрос
    func cancelRequest() {
        currentTask?.cancel()
        currentTask = nil
    }
}
```

currentTask = 3

Исполняется	Очередь запросов
make(request:)	
make(request:)	

Проблемы атомарности

```
actor NetworkService {
    var currentTask: URLSessionDataTask? = nil

    // сделать запрос
    func make(request: URLRequest) async -> Data? {
        let result = await withCheckedContinuation { continuation in
            currentTask = URLSession.shared.dataTask(with: request,
                                                     completionHandler: { data, _, _ in
                continuation.resume(returning: data)
            })
        }
        currentTask = nil
        return result
    }

    // отменить запрос
    func cancelRequest() {
        currentTask?.cancel()
        currentTask = nil
    }
}
```

currentTask = 3

Исполняется	Очередь запросов
make(request:)	
make(request:)	

Проблемы атомарности

```
actor NetworkService {
    var currentTask: URLSessionDataTask? = nil

    // сделать запрос
    func make(request: URLRequest) async -> Data? {
        let result = await withCheckedContinuation { continuation in
            currentTask = URLSession.shared.dataTask(with: request,
                                                     completionHandler: { data, _, _ in
                continuation.resume(returning: data)
            })
        }
        currentTask = nil
        return result
    }

    // отменить запрос
    func cancelRequest() {
        currentTask?.cancel()
        currentTask = nil
    }
}
```

currentTask = 3

Исполняется	Очередь запросов
make(request:)	
make(request:)	
	cancelRequest() - 2

Проблемы атомарности

```
actor NetworkService {
    var currentTask: URLSessionDataTask? = nil

    // сделать запрос
    func make(request: URLRequest) async -> Data? {
        let result = await withCheckedContinuation { continuation in
            currentTask = URLSession.shared.dataTask(with: request,
                                                     completionHandler: { data, _, _ in
                continuation.resume(returning: data)
            })
        }
        currentTask = nil
        return result
    }

    // отменить запрос
    func cancelRequest() {
        currentTask?.cancel()
        currentTask = nil
    }
}
```

currentTask = `nil`

Исполняется	Очередь запросов
make(request:)	
make(request:)	
cancelRequest() - 2	

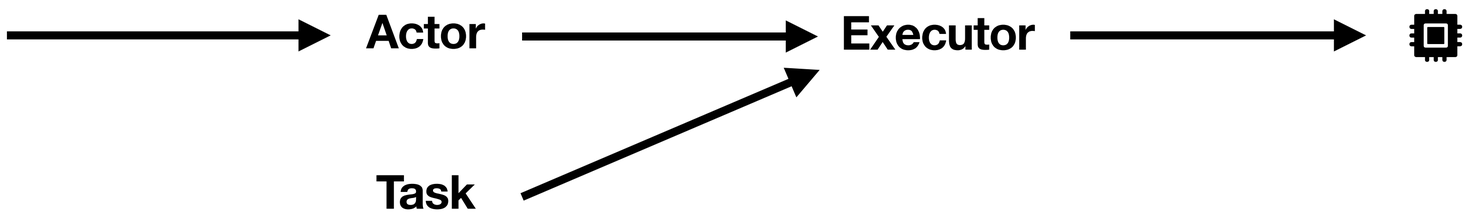
Исполнители
Executors

Исполнители

Actor

Исполнители

Исполнители



Исполнители

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

```swift
protocol SerialExecutor: Executor
```

```swift
@available(SwiftStdlib 6.0, *)
protocol TaskExecutor: Executor
```

Доступные исполнители

Default Actor Executor

```
let actor = MyActor()
await actor.doSomething()
```

Main Actor Executor

```
await MainActor.run {
    // ...
}
```

Global Concurrent Executor

```
Task {
    // ...
}
```

Кастомные исполнители для акторов

Кастомные исполнители для акторов

```swift
final class OperationExecutor: SerialExecutor {
    private let queue: OperationQueue

    init() {
        queue = OperationQueue()
        queue.maxConcurrentOperationCount = 2
    }

    func enqueue(_ job: consuming ExecutorJob) {
        let unownedJob = UnownedJob(job)
        let unownedExecutor = UnownedSerialExecutor(ordinary: self)
        queue.addOperation {
            unownedJob.runSynchronously(on: unownedExecutor)
        }
    }
}
```

```swift
actor Storage {
    private let executor = OperationExecutor()

    nonisolated var unownedExecutor: UnownedSerialExecutor {
        executor.asUnownedSerialExecutor()
    }
}
```

Глобальные акторы

Глобальные акторы

```
public protocol GlobalActor {
  associatedtype ActorType: Actor
  static var shared: ActorType { get }
  static var sharedUnownedExecutor: UnownedSerialExecutor { get }
}
```

```
@globalActor public final actor MainActor: GlobalActor {

  public static let shared = MainActor()

  public nonisolated var unownedExecutor: UnownedSerialExecutor {
      UnownedSerialExecutor(Builtin.buildMainActorExecutorRef())
  }

  public static var sharedUnownedExecutor: UnownedSerialExecutor {
      UnownedSerialExecutor(Builtin.buildMainActorExecutorRef())
  }

  public nonisolated func enqueue(_ job: UnownedJob) {
    _enqueueOnMain(job)
  }
}
```

```
@MainActor
final class MyService {
    // ...
}
```

```
final class MyService {
    @MainActor func doSomething() {
        // ...
    }
}
```

```
func make(work: @MainActor @escaping () -> Void ) {
    // ...
}
```

```
Task {
    // ...
    await MainActor.run {
        // ...
    }
}
```

Создание глобального актора

Создание глобального актора

```
final class NewThreadExecutor: SerialExecutor {
    func enqueue(_ job: consuming ExecutorJob) {
        let unownedJob = UnownedJob(job)
        let unownedExecutor = UnownedSerialExecutor(ordinary: self)
        Thread.detachNewThread {
            unownedJob.runSynchronously(on: unownedExecutor)
        }
    }
}
```

```
@globalActor actor BackgroundActor: GlobalActor {
    static let shared = BackgroundActor()
    private let executor = NewThreadExecutor()
    nonisolated var unownedExecutor: UnownedSerialExecutor {
        executor.asUnownedSerialExecutor()
    }
    static var sharedUnownedExecutor: UnownedSerialExecutor {
        shared.unownedExecutor
    }
}
```

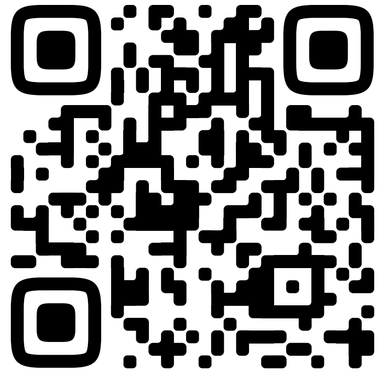
```
@BackgroundActor
final class WorkService {
    func makeWork() {
        // ...
    }
}
```

Распределенные акторы

Distributed actors

Распределенные акторы

```
import Distributed

distributed actor GameService {
    // ...
}
```



Distributed actors и
где они обитают

Наследование контекста

Наследование контекста

```swift
Task {
    // асинхронный контекст
}
```

```swift
Task { @MainActor in
    // асинхронный контекст
}
```

```swift
Task { @MainActor in
    Task {
        // Тут снова MainActor
    }
}
```

```swift
init(
    priority: TaskPriority? = nil,
    @_inheritActorContext @_implicitSelfCapture operation: @Sendable @escaping () async -> Success
  )
```

```swift
static func detached(
    priority: TaskPriority? = nil,
    operation: @Sendable @escaping () async -> Success
  ) -> Task<Success, Failure>
```

А так ли нужны акторы?

А так ли нужны акторы?

Я подозреваю, что пришествие настоящих многоядерных CPU сделает программирование параллельных систем с использованием традиционных мьютексов и разделяемых структур данных сложным до невозможности, и что именно обмен сообщениями станет доминирующим способом разработки параллельных систем.

Джо Армстронг (создателя языка Erlang)