

ВВЕДЕНИЕ В МНОГОПОТОЧНОЕ ПРОГРАММИРОВАНИЕ

ПОТОКИ И ПРОЦЕССЫ



Это процесс?

ПРОЦЕССЫ И ПОТОКИ

Process

Thread 0 Stack

Thread 1 Stack

Thread 2 Stack

Heap

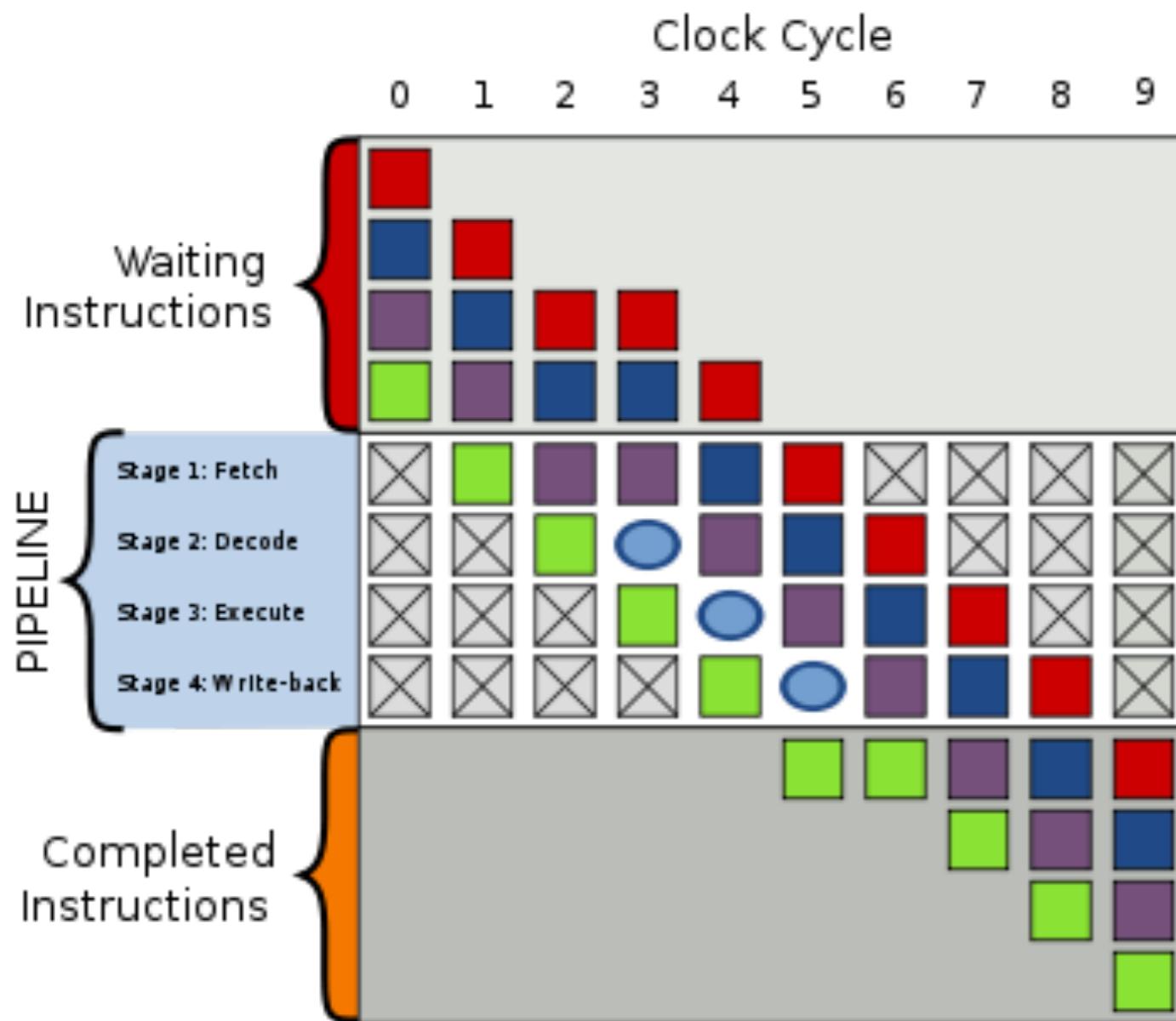
- Процесс - это контейнер для потоков
- У каждого потока свой стек, но общая куча
- А вот у процессов всё свое

КАК ОНИ ВООБЩЕ УСТРОЕНЫ

- зависит от ОС
- потоки в ядре (kernel space)
- пользовательские потоки (user space)
- если вы сами не пишите планировщик, то ваши потоки в kernel space
- так в большинстве современных ОС

КАК УСТРОЕНА МНОГОПОТОЧНОСТЬ

- Несколько ядер или процессоров
- Или на одном процессоре



- конвейерная архитектура процессора
- за один такт выполняется несколько команд
- работает из коробки

**ДАВАЙТЕ ВСЁ РАСПАРАЛЛЕЛИМ
И НАШИ ПРОГРАММЫ
СТАНУТ МОМЕНТАЛЬНЫМИ!**



ЗАКОН АМДАЛА

$$S_p = \frac{1}{\alpha + \frac{1 - \alpha}{p}}$$

- есть p процессоров.
- α - это доля задач, задач могут быть выполнены только последовательно.
- Доля $1 - \alpha$ может быть распараллелена идеально (то есть, чем больше ядер, тем быстрее будет работать).
- Тогда ускорение для задачи, которое получится на системе из p процессоров не будет превышать величину S_p

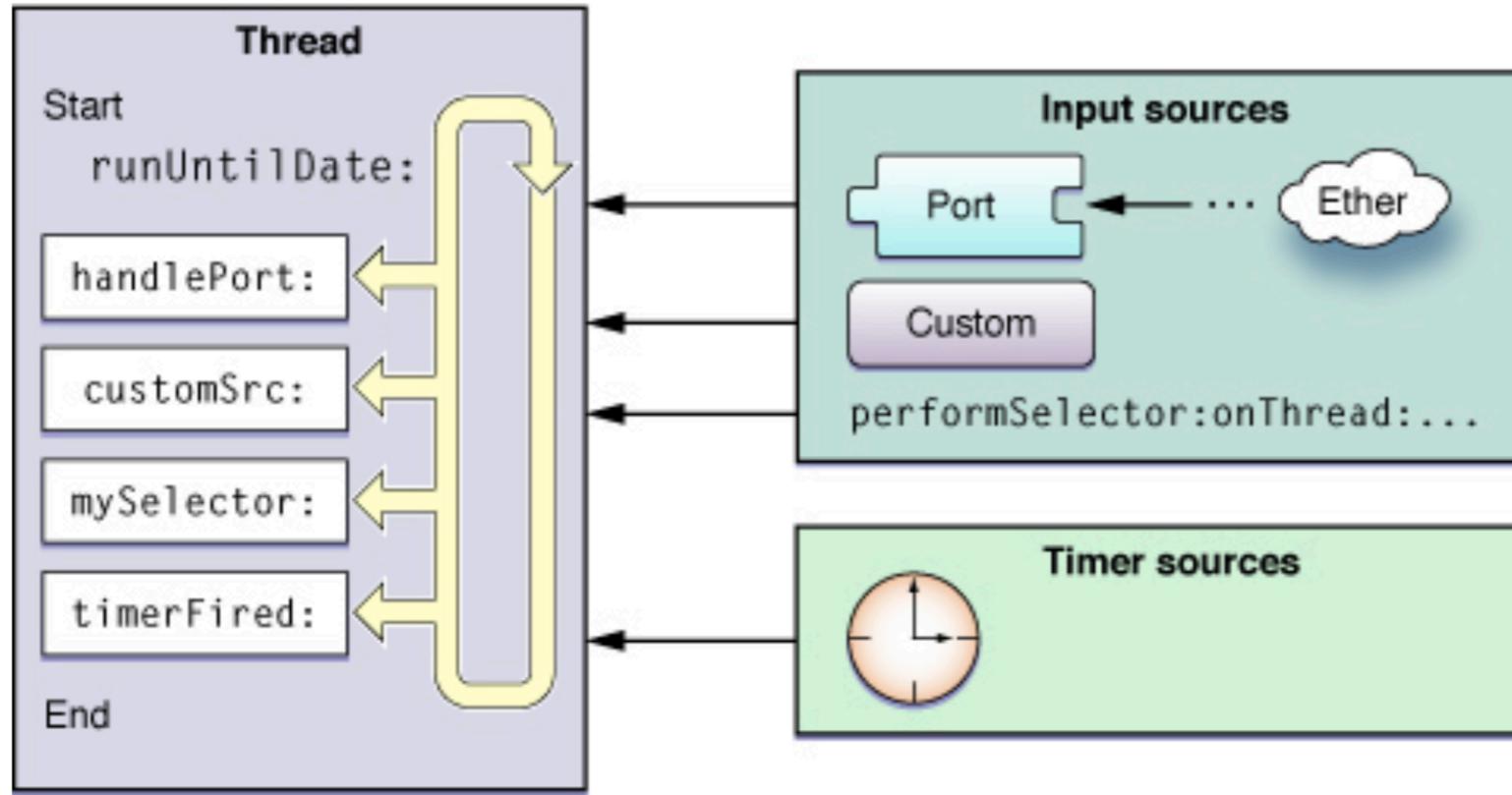
ТАК, И КОГДА ИСПОЛЬЗОВАТЬ ПОТОКИ?

- Задачи на расчеты, которые поддаются распараллеливанию
- При работе с UI или камерой выносим все затратные операции из главного потока
- Работа с вводом-выводом (например, запись в файл, работа с сервером, с базой данных)

МНОГОПОТОЧНОСТЬ В IOS

Несколько уровней

ГЛАВНЫЙ ПОТОК



- цикл обработки событий
- поток находится в цикле ожидания событий
- по приходу события поток запускает обработчик
- Типы событий: инпуты, таймеры, другие источники (performSelector, кастомные)
- RunLoop в iOS >= 1. больше одного. Всегда есть майн. Остальные запускаем самостоятельно
- получается, что главный поток - это тот, на котором работает ранлуп

МЕХАНИЗМЫ РАБОТЫ С МНОГОПОТОЧНОСТЬЮ В IOS

- POSIX-поток (pthread)
- NSThread - iOS-обёртка над pthread
- `performSelectorOnBackground`
- gcd
- NSOperation & NSOperationQueue
- async-await (Swift 5.5, iOS 15+ 😊)

ОЛДСКУЛ - ТРЕДЫ.

PTHREAD

NSThread



```
var nsthread = Thread(block: {
    print("test")
})
nsthread.start()
```

<https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/Multithreading/Introduction/Introduction.html>

КАКОЙ ПОТОК ВАЖНЕЕ? QUALITY OF SERVICE

```
@available(iOS 8.0, *)
public enum QualityOfService : Int {

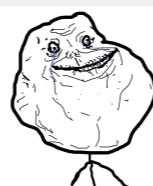
    case userInteractive Самый важный. Обновление интерфейса

    case userInitiated Немедленный результат - по клику пользователя

    case utility Загрузка данных. Баланс энергии, скорости

    case background Скорость совсем не важна

    case `default` По-умолчанию. Между userInitiated и utility
}
```



undefined - для старых апи

```
let thread = Thread {  
    print("test")  
    print(qos_class_self())  
}  
thread.qualityOfService = .userInteractive  
thread.start()
```

GCD

Grand Central Dispatch

- мощный и очень быстрый механизм многопоточности от Apple
- Намного дешевле, чем создание трэдов - добавление задачи в очередь занимает 15 инструкций против нескольких сотен для стандартного потока
- Задачи - функции или блоки

СОСТАВ

- **dispatch_queue** — абстракция очереди задачий, принятых на выполнение. Очередь != thread
- **dispatch_source** — абстракция системных элементов, за которыми мы можем наблюдать: файлы, процессы, порты, таймеры и многое другое.
- **dispatch_data** — обертка над управлением потоками данных: запись / чтение / копирование.

<https://developer.apple.com/documentation/dispatch>

ЕДИНИЦЫ РАБОТЫ

```
23
24      // by block
25      myQueue.async {
26          print("did some work 1")
27      }
28
29      // by work item – gives more control
30      let workItem = DispatchWorkItem {
31          print("did some work 2")
32      }
33
34      myQueue.async(execute: workItem)
35      workItem.cancel()
36
```

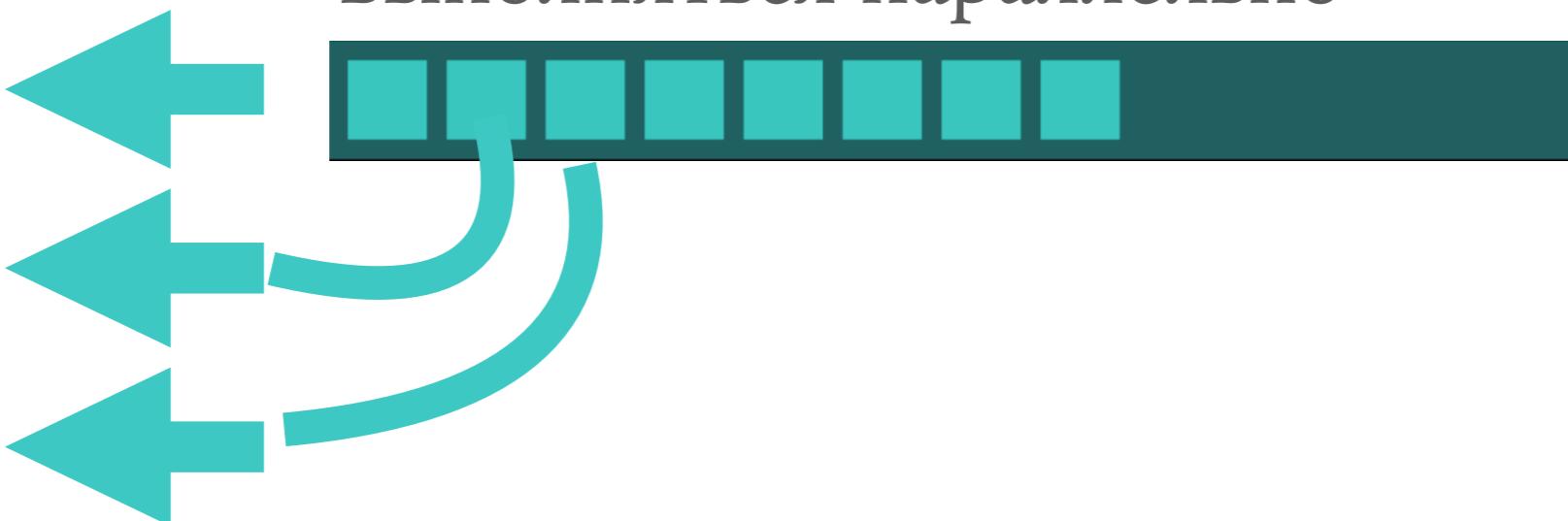
DISPATCH QUEUE

Serial - пока не выполнена задача,
не начнётся другая



```
private let serialQueue = DispatchQueue(label: "serialTest")
```

Concurrent - задачи могут
выполняться параллельно



```
private let concurrentQueue = DispatchQueue(label: "concurrentTest",  
attributes: .concurrent)
```

DISPATCH QUEUE. ВИДЫ ОЧЕРЕДЕЙ

- Global - пул очередей, которые используются системой для своих задач

```
let global = DispatchQueue.global()
```

- Main - главная очередь. В ней весь UI.

```
let main = DispatchQueue.main
```

- Пользовательские

МЕТОДЫ

- `asyncAfter(t)`, t - время - выполняет задачу после другой задачи

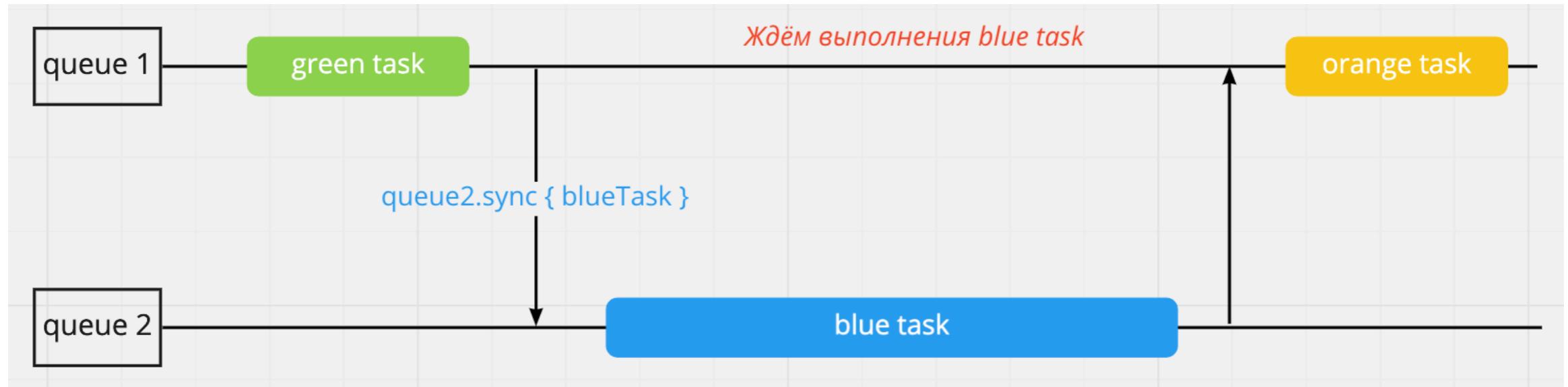
```
DispatchQueue.main.asyncAfter(deadline: .now() + 3) {  
    print("Dispatching after")  
}
```

```
DispatchQueue.main.asyncAfter(deadline: .now() + 3, qos: .userInteractive, flags: []) {  
    print("Dispatching after")  
}
```

- Выполнить задачу синхронно `.sync()`
- Выполнить задачу асинхронно `.async()`

СИНХРОННОЕ ВЫПОЛНЕНИЕ

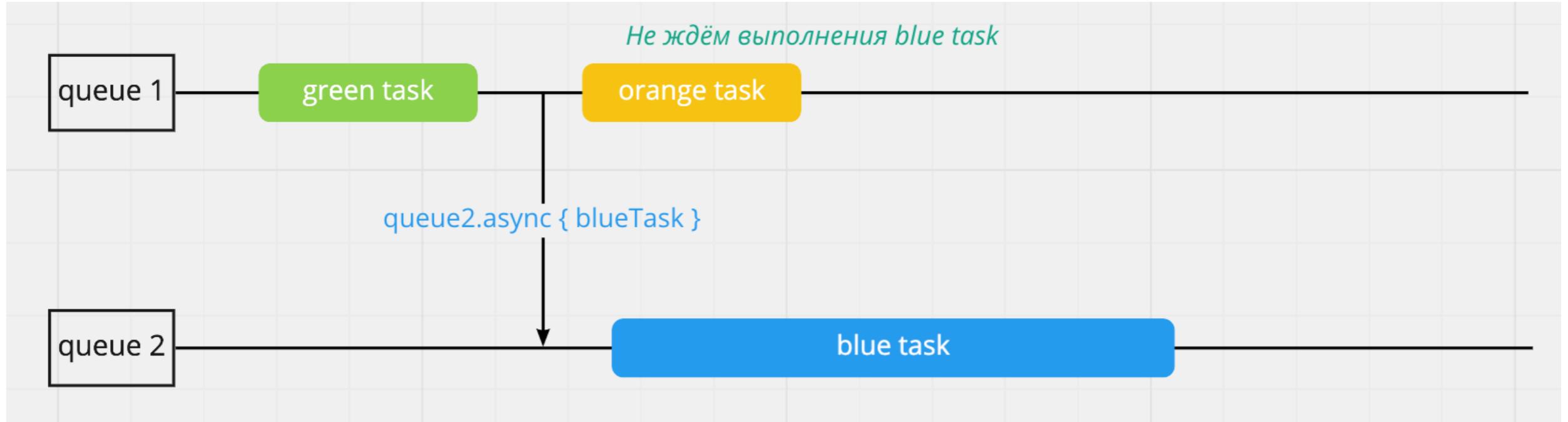
```
queue.sync {  
    // slow non-UI task  
    print("sync")  
}
```



Если другая очередь занята другой задачей - ждём, пока освободится, кладём нашу задачу и, когда она сделается, можем делать следующие.

АСИНХРОННОЕ ВЫПОЛНЕНИЕ

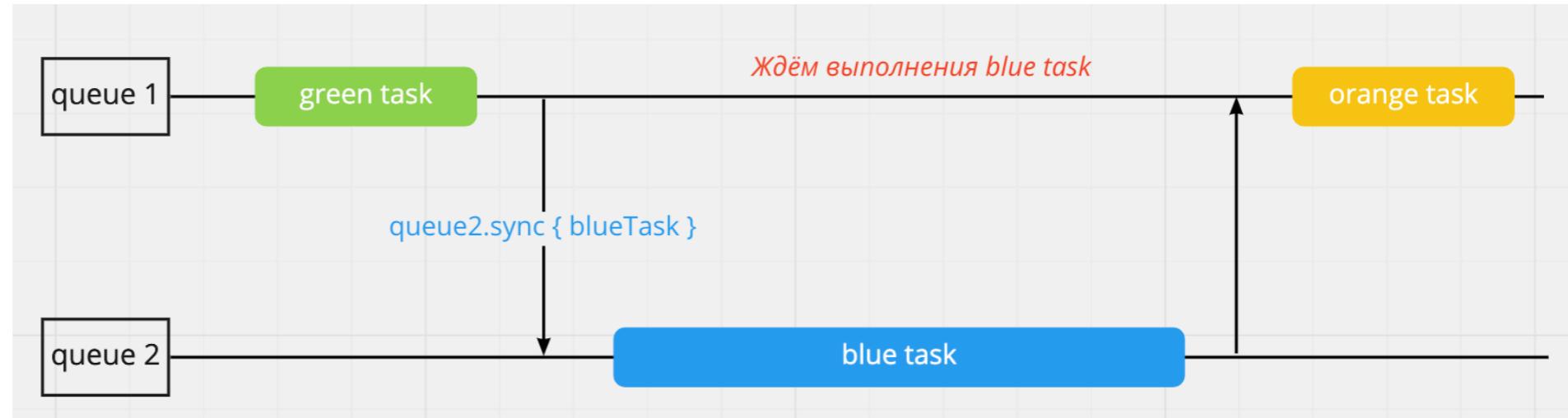
```
queue.async {  
    // slow non-UI task  
    print("async")  
}
```



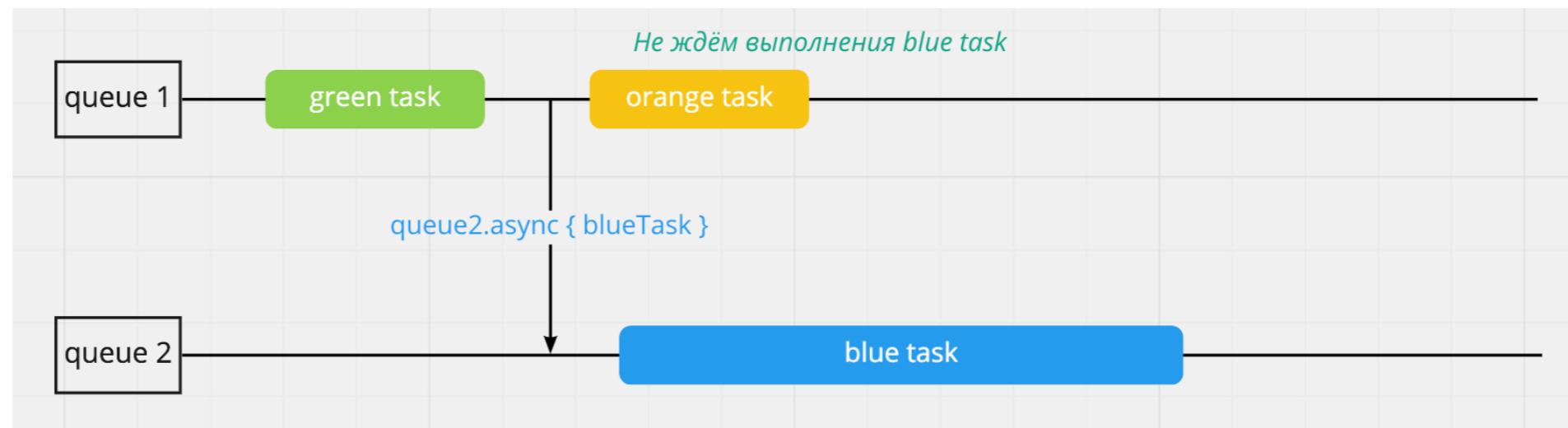
Что происходит, если Другая очередь занята?
Закидываем туда задачу, нам всё равно, когда она начнётся.

ЕЩЁ РАЗОК!

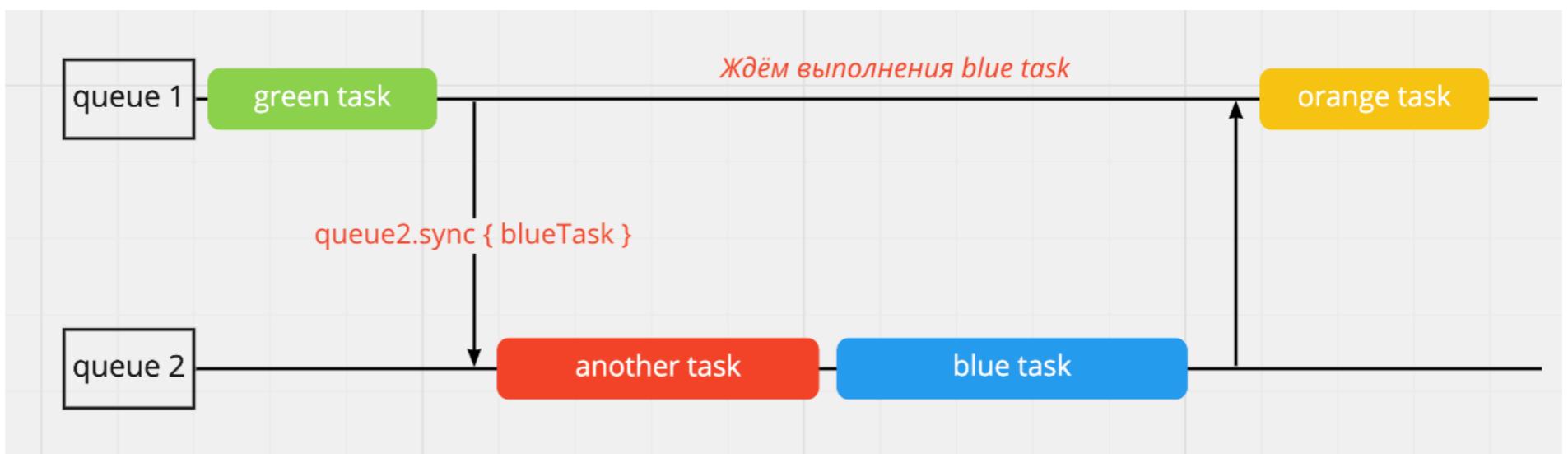
```
queue.sync {  
    print("sync")  
}
```



```
queue.async {  
    print("async")  
}
```



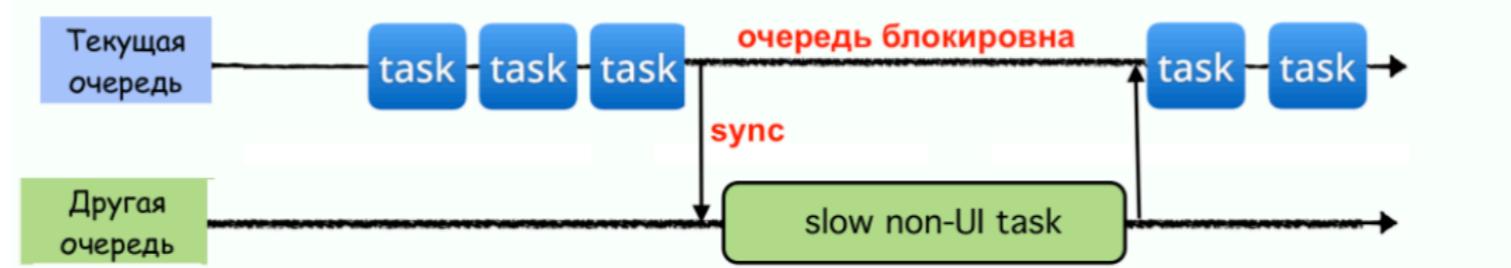
Note that



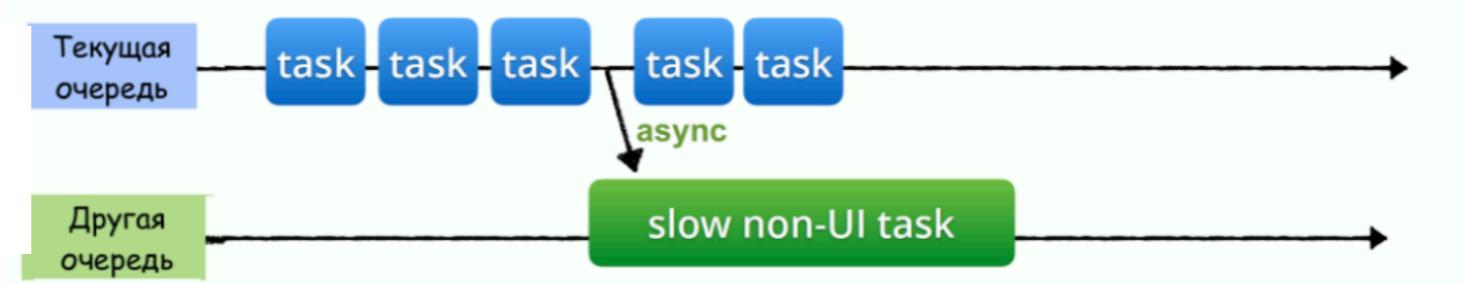
ТРЕНИРУЕМСЯ!

ЧТО ВЫВЕДЕТСЯ НА ЭКРАН?

```
queue.sync {  
    print("sync")  
}
```



```
queue.async {  
    print("async")  
}
```



```
let concurrentQueue = DispatchQueue(label: "concurrentQueue", attributes: .concurrent)  
concurrentQueue.sync {  
    print("test1")  
}  
print("test2")
```

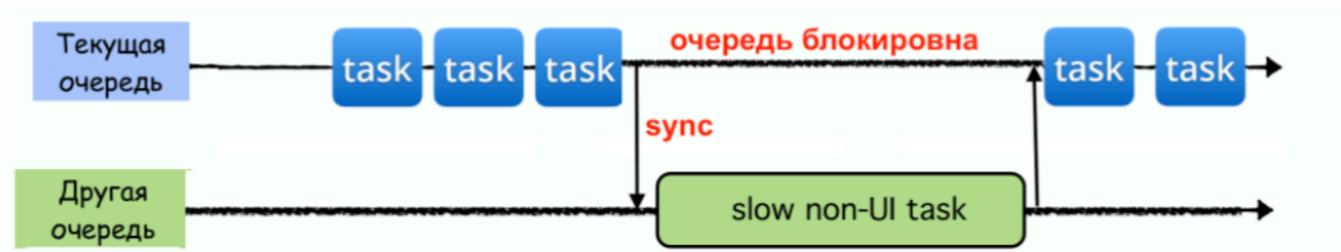
test1 test2

```
serialQueue.sync {  
    print("test1")  
}  
serialQueue.async {  
    print("test2")  
}  
serialQueue.sync {  
    print("test3")  
}
```

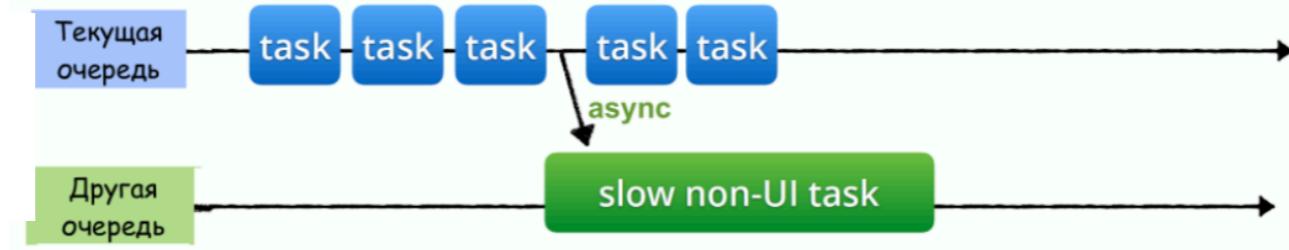
test1 test2 test3

ЧТО ВЫВЕДЕТСЯ НА ЭКРАН?

```
queue.sync {  
    print("sync")  
}
```



```
queue.async {  
    print("async")  
}
```



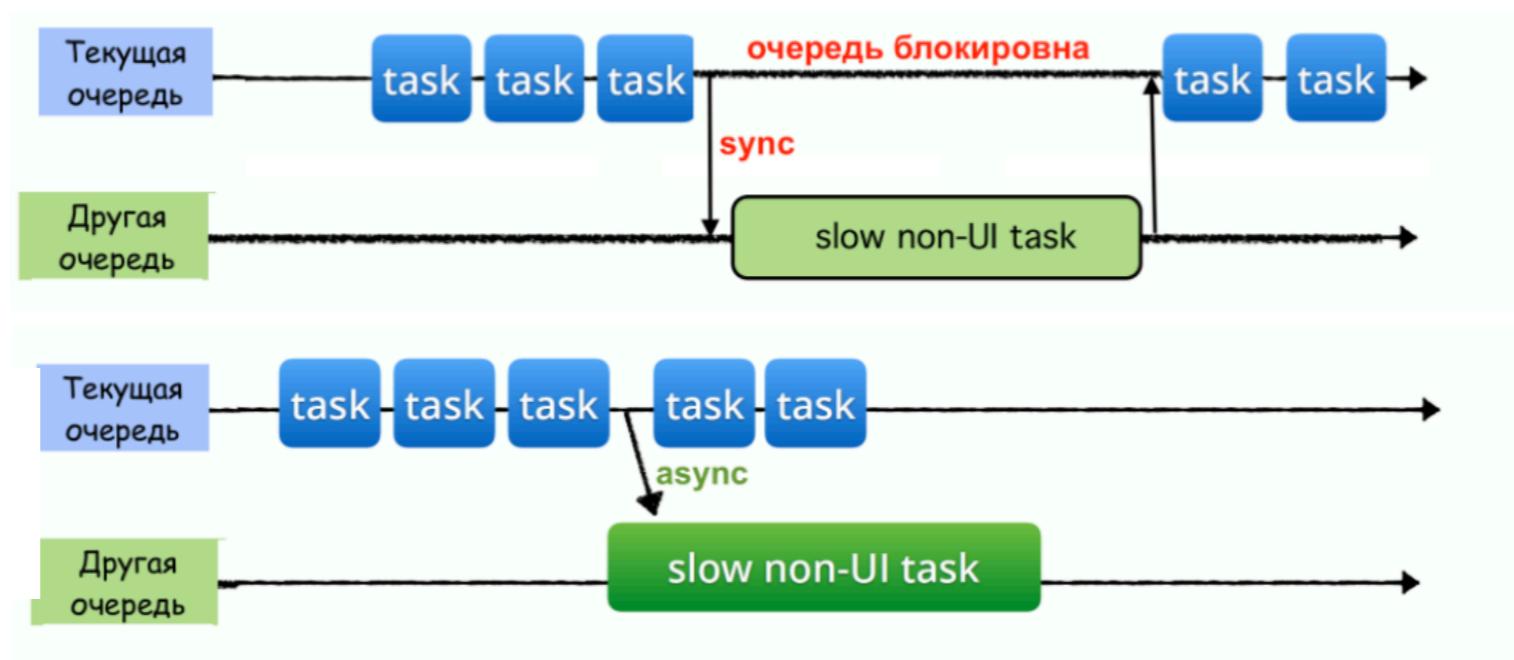
```
let serialQueue = DispatchQueue(label: "serial")  
serialQueue.sync {  
    sleep(1)  
    print("test1")  
}  
print("test2")
```

test1, test2

```
let serialQueue = DispatchQueue(label: "serialQueue")  
serialQueue.async {  
    print("test1")  
}  
print("test2")
```

Возможны оба варианта

```
queue.sync {
    print("sync")
}
```



```
queue.async {
    print("async")
}
```

```
class AsyncVsSyncTest1 {
    private let serialQueue = DispatchQueue(label:
"serialTest")

    func testSerial() {
        serialQueue.async {
            print("test1")
        }

        serialQueue.async {
            sleep(1)
            print("test2")
        }

        serialQueue.sync {
            print("test3")
        }

        serialQueue.sync {
            print("test4")
        }
    }
}
```

Serial:
test1
test2
test3
test4

```
class AsyncVsSyncTest2 {
    private let concurrentQueue = DispatchQueue.global()

    func testConcurrent() {
        concurrentQueue.async {
            print("test1")
        }

        concurrentQueue.async {
            print("test2")
        }

        concurrentQueue.sync {
            print("test3")
        }

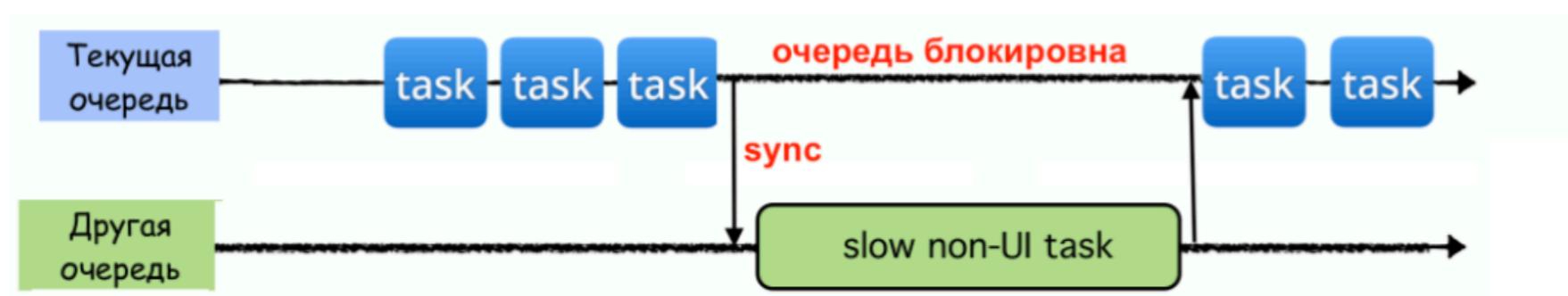
        concurrentQueue.sync {
            print("test4")
        }
    }
}
```

Concurrent:
test3 -> test4

ДЭДЛОКИ

```
let firstQueue = DispatchQueue(label: "first queue")
let secondQueue = DispatchQueue(label: "second queue")
```

```
18    firstQueue.sync {
19        print("first 1")
20
21        secondQueue.sync {
22            print("second 1") ≡ Thread 1: EXC_BAD_INSTRUCTION
23
24            firstQueue.sync {
25                print("first 2")
26            }
27
28            print("second 2")
29        }
30        print("first 3")
31    }
```



```
override func viewDidLoad() {
    super.viewDidLoad()

    DispatchQueue.main.sync { ≡ Thread 1: EXC_BAD_INSTRUCTION
        print("hi")
    }
```

```
override func viewDidLoad() {
    super.viewDidLoad()
```

```
firstQueue.sync {
    print("first 1")
```

```
DispatchQueue.main.sync { ≡
    print("hi")
```

```
}
```

```
}
```



OPERATIONS

NSOPERATIONQUEUE (OPERATIONQUEUE)

Operation

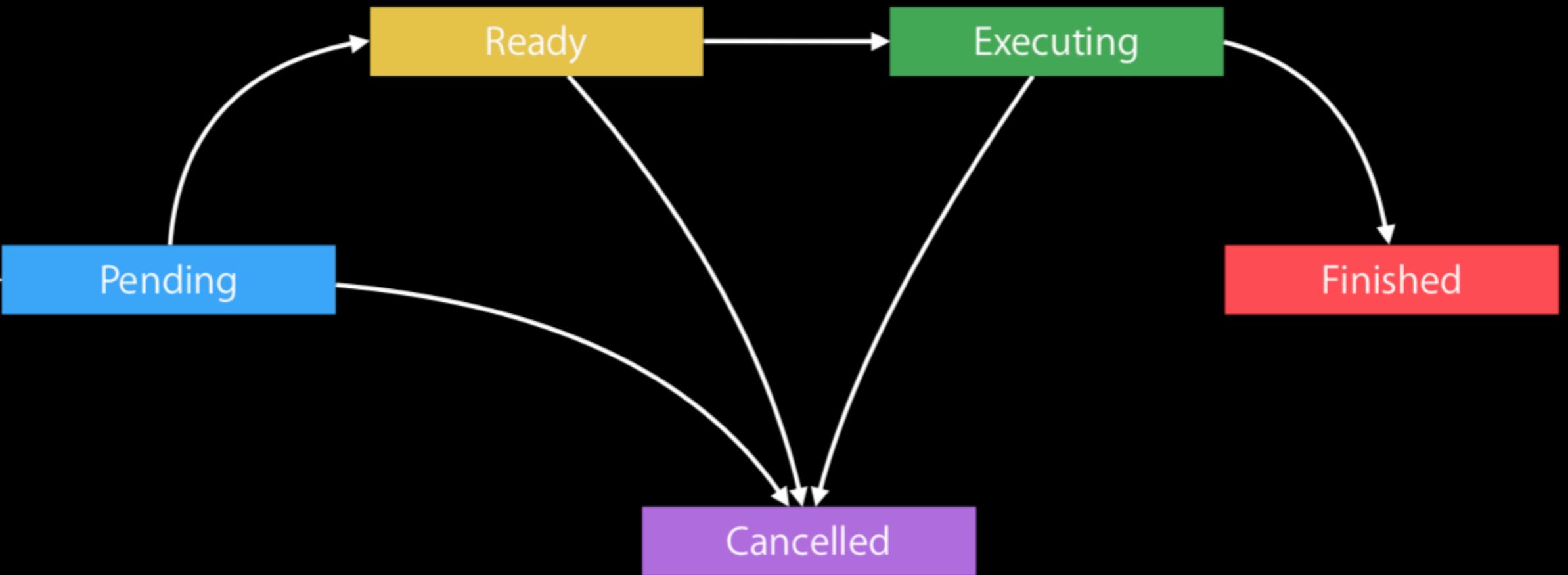


OperationQueue

maxConcurrentOperationCount

Serial
Concurrent

ЖИЗНЕННЫЙ ЦИКЛ ОПЕРАЦИИ



Можно наблюдать за ними через KVO

ИСПОЛЬЗОВАНИЕ

Блоковая

```
private let operationQueue = OperationQueue()

func test() {
    let blockOperation = BlockOperation {
        print("test")
    }
    operationQueue.addOperation(blockOperation)
}
```

Наследование

```
9
10 class FirstOperation: Operation {
11
12     override func main() {
13         print("first operation running")
14         sleep(2)
15
16         if isCancelled {
17             print("first operation was cancelled")
18             return
19         }
20
21         print("first operation still running")
22
23     }
24 }
```

Вызывается

```
15     let operationsQueue = OperationQueue()
16
17     override func viewDidLoad() {
18         super.viewDidLoad()
19
20         let operation = FirstOperation()
21         operationsQueue.addOperation(operation)
22         print("c")
23         operation.cancel()
24     }
```

ЗАВИСИМОСТИ И КОМПЛИШН БЛОКИ

```
let networkingOperation = NetworkingOperation()
let resizingOperation = FirstOperation()

resizingOperation.addDependency(networkingOperation)

operationsQueue.addOperations([networkingOperation, resizingOperation], waitUntilFinished: true)
```

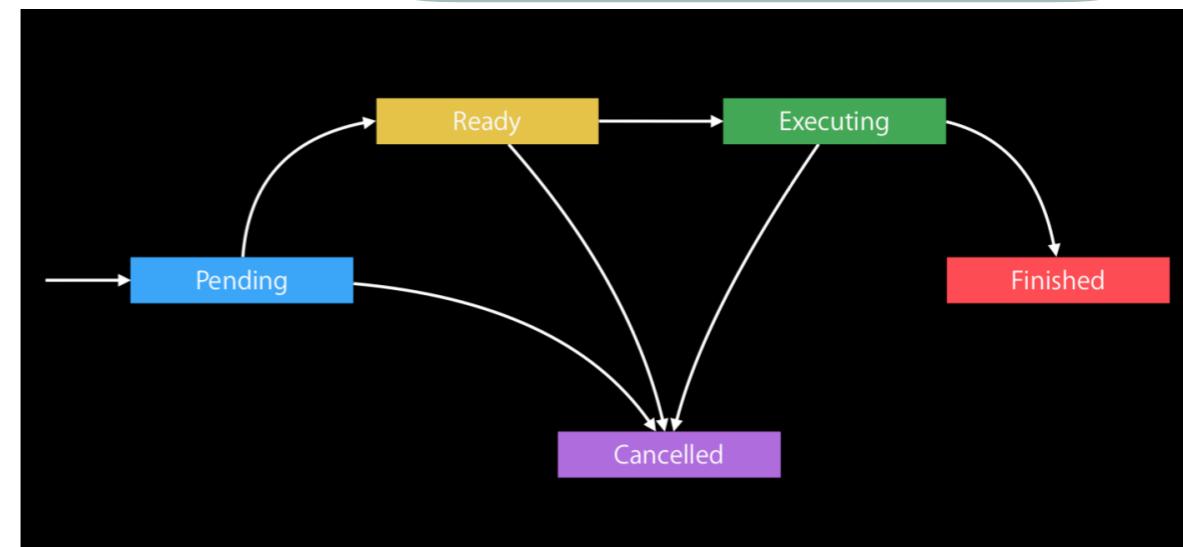
или

```
25    let networkingOperation = NetworkingOperation()
26
27    networkingOperation.completionBlock = {
28        print("hurray!")
29    }
30
31    operationsQueue.addOperation(networkingOperation)
.
```

Операция, запускающая внутри себя асинхронный блок...

```
10 class NetworkingOperation: Operation {
11
12     public override var isAsynchronous: Bool {
13         return true
14     }
15
16     public override var isExecuting: Bool {
17         return state == .executing
18     }
19
20     public override var isFinished: Bool {
21         return state == .finished
22     }
23
24 // MARK: - State management
25
26     public enum State: String {
27         case ready = "Ready"
28         case executing = "Executing"
29         case finished = "Finished"
30         fileprivate var keyPath: String { return "is" + self.rawValue }
31     }
32
33     /// Thread-safe computed state value
34     public var state: State {
35         get {
36             stateQueue.sync {
37                 return stateStore
38             }
39         }
40         set {
41             let oldValue = state
42             willChangeValue(forKey: state.keyPath)
43             willChangeValue(forKey: newValue.keyPath)
44             stateQueue.sync(flags: .barrier) {
45                 stateStore = newValue
46             }
47             didChangeValue(forKey: state.keyPath)
48             didChangeValue(forKey: oldValue.keyPath)
49         }
50     }
51
52     private var stateStore: State = .ready
53
54     private let stateQueue = DispatchQueue(label: "AsynchronousOperation State Queue")
55
56     override func main() {
57         print("networking started")
58         DispatchQueue.global().async {
59             sleep(2)
60             print("networking finished")
61             self.state = .finished
62         }
63     }
64 }
```

Никакой магии, это всё KVO
вокруг стейтов



**HTTPS://DEVELOPER.APPLE.COM/DOCUMENTATION/FOUNDATION/
NSOPERATION**

ASYNC/AWAIT

ЛИТЕРАТУРА

-  <https://developer.apple.com/wwdc15/226> - видос от эппл про NSOperations
- <https://webnewsite.ru/mnogopotochnost-v-ios-vvedenie-v-gcd/> статья про многопоточность в айос на русском
- <https://www.raywenderlich.com/5370-grand-central-dispatch-tutorial-for-swift-4-part-1-2> - туториал по GCD от Рэя
- <https://swiftbook.ru/post/tutorials/tutorial-po-grand-central-dispatch-dlya-swift-chast-22/> - он же, на русском
-  <https://stepik.org/lesson/51634/step/1?unit=29896> - курс на стекипке по многопоточности (на русском)
-  Глава про многопоточность в Силе Objective-C
- <https://www.raywenderlich.com/3648-ios-concurrency-with-gcd-and-operations/lessons/1> - курс от raywenderlich