

# Разработка под iOS. Взлетаем

Часть 1

## Многопоточность

# Оглавление

<b>1</b>	<b>Разработка под iOS. Взлетаем .....</b>	<b>3</b>
1.1.	Введение .....	3
1.2.	GCD .....	6
1.3.	Операции .....	13

# Глава 2

## 1 Разработка под iOS. Взлетаем

### 1.1. Введение

Мы начинаем вторую часть курса "Разработка под iOS", и первой темой в этой части будет многопоточность.

Открывать вторую часть курса выпала честь мне. Меня зовут Дима, в Яндексе я делаю приложение, которое называется «Яндекс».

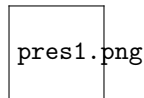
Залогом хорошего User experience для мобильных приложений является отзывчивый UI. Люди давно начали думать, какими способами этого можно добиться.

Мы начнем с того, что очень быстро посмотрим на самые базовые вещи в многопоточности; затем разберемся с технологией Grand Central Dispatch, или GCD; и, конечно, мы узнаем, что такое операции, ведь они будут в домашнем задании.

Итак, за 12 лет до первого айфона, в 1995 году, свет увидел стандарт под номером IEEE Std 1003.1c-1995, описывающий POSIX потоки, или PThreads.

Стандарт стал так распространен, что его реализации существуют под все операционные системы, включая Linux, Windows, macOS и конечно iOS.

**Потоки** - основа выполнения приложения. Каждое приложение состоит как минимум из одного процесса, а процесс состоит хотя бы из одного потока - главного потока.



Но у процесса могут быть и другие потоки. Все потоки разделяют общие ресурсы процесса, такие как память, файловые дескрипторы и процессорное время. В то же время, у каждого потока есть свой **стек** - область памяти, в которой, пока функция выполняется, сохраняются её локальные переменные.

Грамотная организация работы с потоками позволяет сделать ваше приложение быстрым, предотвратить зависающий UI и реагировать на действия пользователя без промедления. Кроме того, потоки нужны нам, чтобы параллельно обрабатывать данные. Разумеется, ограничения физического мира таковы, что мы не можем выполнять действительно одновременно больше потоков, чем ядер у нашего процессора. Что мы действительно имеем в виду, говоря, что эти потоки будут выполняться одновременно - то, что операционная система с помощью специального механизма - планировщика будет быстро-быстро переключать конкретное ядро процессора на выполнение то одного потока, то другого, то третьего, таким образом создавая иллюзию, что потоки выполняются одновременно.

**pthread** - это самый низкоуровневый инструмент, доступный в приложении. Он дает почти прямой доступ к управлению потоками ядра ОС. Среди его возможностей: создание потоков, управление их приоритетами, а также предоставление различных механизмов синхронизации.

```
pthread_create()  
pthread_exit()  
pthread_cancel()
```

```
pthread_join()
pthread_detach()
pthread_attr_init()
pthread_attr_setdetachstate()
pthread_attr_destroy()
pthread_mutex_*(())
pthread_cond_*(())
```

Но эта библиотека сложна в использовании, особенно в Swift'e, из-за необходимости работы с указателями. К счастью, в iOS SDK существует более простой способ управления потоками - класс Thread.

```
let thread = Thread {
    // вычисления
}
thread.start()
```

Конструктор класса Thread принимает аргументом блок, который будет исполняться этим потоком. Но создать поток недостаточно. Чтобы его запустить, еще необходимо вызвать метод start у объекта потока. Только после этого код, написанный в блоке, начнет исполняться.

Разумеется, классом Thread все не заканчивается. При написании многопоточных приложений почти всегда требуется работать с общими данными из разных потоков. А это означает, что потоки нужно синхронизировать. Даже для того, чтобы в одном потоке просто дождаться, когда завершится выполнение другого, нужна синхронизация. Для синхронизации потоков существуют объекты синхронизации. К таким объектам относится Mutex, который в iOS SDK представлен классами NSLock и NSRecursiveLock.

Посмотрим, как работать с NSLock, на таком примере.

```
var counter = 0
let thread1 = Thread {
    for _ in 0..<1000 {
        counter += 1
    }
}
let thread2 = Thread {
    for _ in 0..<1000 {
        counter += 1
    }
}
thread1.start()
thread2.start()
// counter < 2000
```

Два потока одновременно увеличивают какую-то переменную - счетчик - на единицу, каждый по тысяче раз. Ожидаемый результат после выполнения обоих циклов - число 2000 в переменной counter. Однако в таком варианте ее значение будет всегда меньше.

Дело в том, что операция увеличения счетчика не атомарна. Она состоит из нескольких шагов, а именно: чтение уже хранящегося в переменной значения, сложение этого значения с единицей и запись результата обратно в переменную.

```
counter+=1
```

Эквивалентно

```
let temp = counter + 1
counter = temp
```

В условиях отсутствия синхронизации между потоками первый и второй поток могут одновременно оказаться в точке кода, производящей чтение. Например, так:

поток 1 считывает значение переменной, которое в этот момент равняется нулю, прибавляет единицу, и получает единицу в качестве результата:

```
counter==0
```

поток 2 считывает значение переменной, и оно так же равняется нулю, поскольку первый поток не успел записать новое значение переменной:

```
counter==0
```

далее выполняется сложение, результат которого - также единица.

Вот здесь и кроется проблема.

Первый поток продолжает выполнение и записывает новое значение, то есть единичку, в переменную:

```
counter==1
```

Затем и второй поток делает то же самое. И в результате значение счетчика увеличилось только на 1:

```
counter==1 // а не 2
```

Сделать этот код корректным очень легко. Нужно синхронизовать обращение к счетчику.

Для этого мы создаем экземпляр класса `NSLock`:

```
let lock = NSLock()
var counter = 0
let thread1 = Thread {
    for _ in 0..<1000 {
        lock.lock()
        counter += 1
        lock.unlock()
    }
}
let thread2 = Thread {
    for _ in 0..<1000 {
        lock.lock()
        counter += 1
        lock.unlock()
    }
}
thread1.start()
thread2.start()
```

Затем в коде потоков вызываем у него метод `lock` перед операцией изменения счетчика, и `unlock` после. Участок кода между `lock` и `unlock` называется **критической секцией**.

Теперь выполнение наших потоков будет происходить следующим образом. Оба потока попытаются захватить мьютекс, вызывая функцию `lock`, но первому потоку это удастся успешно. А вот второй поток блокируется, и будет заблокирован все время, пока первый поток не освободит мьютекс вызовом функции `unlock`. Затем первый поток выполняет чтение переменной, сложение и запись обратно:

```
counter==1
```

затем вызывает функцию `unlock` у объекта класса `NSLock`, и в этот момент разблокируется выполнение второго потока. Теперь мьютекс захвачен уже им. Далее второй поток также выполняет увеличение счетчика:

```
counter==2
```

А в это время первый поток совершает цикл, вновь оказавшись в точке вызова функции `lock`. Но, поскольку мьютекс сейчас захвачен вторым потоком, первый поток блокируется.

Затем второй поток заканчивает увеличение счетчика, освобождает мьютекс и владение мьютексом переходит снова к первому потоку. В такой схеме работы мы не разрешаем потокам одновременно выполнять нашу критическую секцию, не разрешаем одновременно увеличивать счетчик. Поэтому и результат будет правильным.

В данном примере использован класс `NSLock`. Он устроен так, что разрешает вызывать `unlock` только тому потоку, который до этого вызвал `lock`. Есть также такое ограничение, что один и тот же поток не может два раза подряд вызвать `lock`. Это приведет к `deadlock`-у, то есть ситуации, когда вообще невозможно продолжить выполнение программы.

```
let lock = NSLock()
lock.lock()
lock.lock() // deadlock
```

Но еще есть класс `NSRecursiveLock`, который как раз позволяет одному и тому же потоку вызывать `lock` много раз подряд. При этом для того, чтобы его освободить, нужно вызвать `unlock` такое же количество раз.

```
let lock = NSRecursiveLock()
lock.lock()
lock.lock()
lock.unlock()
lock.unlock()
```

Кроме мьютексов, есть и другие объекты синхронизации:

- `NSConditionLock` и `NSCondition`, реализующие синхронизацию с помощью так называемых условий;
- устаревший `OSSpinLock` и `os_unfair_lock` - работает как обычный мьютекс, только быстрее, но может впустую тратить процессорное время;
- а вот класса `NSSemaphore` нет. Вместо этого предлагается пользоваться либо низкоуровневым POSIX API, а точнее типом `sem_t`, либо классами обозначенными выше, либо более высокоуровневыми инструментами.

Именно о них мы поговорим в следующей части.

## 1.2. GCD

.

2009 год. В прокат выходит фильм Аватар, на каждом втором смартфоне установлена Angry Birds, а Apple выпускает обновление для своей десктопной операционной системы Mac OS X Snow Leopard. Именно в этом релизе появляется Grand Central Dispatch, или GCD. Это технология для управления многопоточностью на базе паттерна «пул потоков». Это означает, что вместо того, чтобы программист сам создавал и управлял потоками, за него это делает система.

GCD вводит понятие очереди исполнения. Она представлена классом **DispatchQueue**.

```
let queue = DispatchQueue.global()
queue.async(execute: {
    // код выполняющийся в фоне
})
```

**Очередь** - это список задач, которые необходимо выполнить. А задачи - это фактически сам код.

Так выглядит запуск задачи в фоновом потоке:

Обращаясь к статическому методу `global()` у класса `DispatchQueue`, мы получим объект очереди, управляющей потоками фонового выполнения. Затем, вызвав у этого объекта метод `async`, передав аргументом блок с нашим кодом, мы создадим в этой очереди задачу на исполнение этого блока и сразу же продолжим выполнение кода после вызова `async`. А фактически наша задача начнет выполняться, когда в этой очереди найдется свободный поток.

Но что, если мы захотим выполнить код на другом потоке, но дождаться, когда он точно завершится? Что ж, для этого есть метод `sync`.

Продемонстрировать это можно так.

Рассмотрим первый фрагмент кода:

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

и второй фрагмент кода:

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

Оба фрагмента печатают два слова из двух разных потоков - Hello и world.

В первом примере код

```
print("Hello")
```

ставится в очередь с помощью метода `sync`, поэтому, пока он не выполнится, второй `print` (со словом World) даже не будет вызван. В результате вывод этого кода всегда будет одинаковым:

```
Hello World
```

А во втором примере код

```
print("Hello")
```

ставится в очередь методом `async`, который не выполняет ожидание. Поэтому порядок слов в выводе не гарантируется: они будут появляться в случайном порядке.

Метод `sync` является не единственным способом синхронизации задач в GCD. Но перед тем, как углубляться в детали синхронизации, вернемся к началу.

Очереди бывают двух типов - Serial и Concurrent.

Задачи в очереди типа Serial всегда выполняются одна за другой на одном потоке, а задачи в очереди типа Concurrent распределяются одновременно по нескольким потокам и выполняются «параллельно».

Слово «параллельно» взято в кавычки, потому что, как мы помним, на самом деле наша параллельность ограничена числом ядер процессора. Но в данном контексте важно, что в отличие от Serial задачи в Concurrent не дожидаются друг друга, а начинают выполняться, как только для выполнения этой задачи будет найден свободный поток.

Примером Serial очереди является главная очередь приложения. Эта очередь очень особенная, потому что она ассоциирована с главным потоком приложения. Именно на этой очереди работает весь UI приложения, все анимации и реакции на ввод пользователя.

Получить доступ к главной очереди можно с помощью статического свойства `main` у класса `DispatchQueue`:

```
DispatchQueue.main.async {  
    label.text="Hello World"  
}
```

Подобные конструкции можно нередко встретить в коде iOS приложений. Дело в том, что работать с UI элементами можно только из главного потока. В противном случае мы можем получить ошибку или, что хуже, неконсистентное состояние приложения. Поэтому каждый раз, получив в фоне данные из базы данных или из сети, необходимо переключаться на главный поток, чтобы их отобразить в UI. Какие еще очереди бывают, кроме главной? GCD предоставляет разработчику несколько фоновых глобальных Concurrent очередей, которые различаются параметром QoS или Quality of Service.

Задается в виде

```
DispatchQueue.global(qos: xxx)
```

Параметр `xxx` напрямую влияет на то, какой приоритет будет у задач этой очереди, и может принимать одно из следующих значений:

`.userInteractive`. Подходит для задач, которые взаимодействуют с пользователем в данный момент и результат которых требуется получить как можно быстрее. Задачи, откладывание которых приведет к видимым для пользователя лагам. Например, обработка изображения с камеры в реальном времени. Наивысший приоритет.

`.userInitiated`. Для задач, которые запустил пользователь и необходимо дождаться результата выполнения этих задач. Например обновление данных после того, как пользователь сделал Pull to refresh. Такие задачи могут занимать несколько секунд. То есть задачи, которые должны выполняться быстро, но не мгновенно.

Приоритет высокий, но ниже, чем у `.userInteractive`.

`.utility`. Подходит для случаев, когда пользователь напрямую не запрашивал выполнение какой-то задачи и ее выполнение можно отложить.

Имеет приоритет ниже, чем `.userInitiated`.

`.background`. Используется, когда время выполнения задачи совсем не критично.

Имеет наиболее низкий приоритет.

Также есть глобальная очередь без указания QoS:

```
● DispatchQueue.global()
```

Задачи этой очереди имеют класс обслуживания, равный `default`, который по приоритету находится между `.userInitiated` и `.utility`.



Кроме этих глобальных очередей, в GCD мы можем создать отдельную очередь для своих каких-то нужд. Например, отдельная очередь для операций парсинга ответов сервера или очередь для обработки картинок.

Для этого класс `DispatchQueue` предоставляет конструктор, который принимает параметры `label`, `QoS` и `attributes` - соответственно имя очереди, используемое при отладке, значение класса обслуживания и список атрибутов, с помощью которых, например, можно указать, что очередь должна быть `concurrent`. А также параметры `autoreleaseFrequency` и `target`, которые мы рассматривать не будем (их можно просто не указывать при вызове этого конструктора).

```
let demoSerialQueue = DispatchQueue(
    label: "demo serial queue",
    qos: .utility
)
let demoConcurrentQueue = DispatchQueue(
    label: "demo concurrent queue",
    qos: .utility,
    attributes: [.concurrent]
)
```

Результатом вызова этого конструктора будет объект очереди, с которым можно взаимодействовать так же, как и с глобальными очередями. Но к этой очереди нельзя будет никак обратиться, кроме как через этот объект.

Иногда бывает нужно синхронизировать выполнение нескольких задач. Для этого могут быть полезными классы `DispatchSemaphore` и `DispatchGroup`. `DispatchSemaphore` реализует, как не сложно догадаться, объект синхронизации Семафор и позволяет ограничить число одновременно выполняемых задач.

Чтобы понять, как он работает, разберем такой пример. Создаем семафор с начальным значением 2. Размещаем на глобальной очереди 4 задачи, каждая из которых вызывает `wait` сразу после входа и `signal` перед выходом. А между этими двумя вызовами выполняется какой-то полезный код.

```
let semaphore = DispatchSemaphore(value: 2)
for i in 0..<4 {
    DispatchQueue.global().async{
        semaphore.wait()
        sleep(1)
        semaphore.signal()
    }
}
```

Далее можно представить, что у нас появилось 4 потока:

```
Thread1:
{
    semaphore.wait()
    sleep(1)
    semaphore.signal()
}

Thread2:
{
    semaphore.wait()
    sleep(1)
```

```

    semaphore.signal()
}

Thread3:
{
    semaphore.wait()
    sleep(1)
    semaphore.signal()
}

Thread4:
{
    semaphore.wait()
    sleep(1)
    semaphore.signal()
}

```

Они все сейчас в начальной точке выполнения, а счетчик семафора равен двум:

```
Semaphore counter == 2
```

Затем один из потоков вызывает функцию `wait`. Мы не можем заранее сказать, который из потоков окажется первым, ведь мы размещали эти задачи на concurrent очереди, но для простоты пронумеруем эти потоки от 1 до 4 и будем их так называть. Вызов `wait` уменьшает значение семафора на 1:

```
Semaphore counter == 1
```

Затем второй поток также вызывает `wait`. В этот момент значение счетчика становится равным нулю:

```
Semaphore counter == 0
```

Последующие вызовы `wait` будут блокировать выполнение. Поэтому потоки 3 и 4 оказываются заблокированы.

Далее первый поток выполняет какой-то полезный код. Второй поток занят тем же самым.

А затем первый поток, заканчивая свое выполнение, вызывает функцию `signal`, которая увеличивает значение семафора на 1. Это разблокирует третий поток, и теперь уже он снова уменьшает значение семафора и начнет выполнять свой полезный код.

Затем поступает сигнал от второго потока, семафор снова увеличивается на 1, разблокируется 4й поток и семафор снова становится равен нулю.

Далее выполняются 3й и 4й потоки.

И когда они оба вызовут `signal`, счетчик будет равен изначальному значению, то есть 2.

Таким образом мы ограничили одновременное выполнение до двух потоков.

`DispatchGroup` позволяет реализовать ожидание основанное на счетчике, который можно увеличивать и уменьшать с разных потоков.

Давайте посмотрим на такой пример. Он выводит слово `Hello` с одного потока, а `World` с другого и только в правильном порядке.

```

let group = DispatchGroup()
group.enter()
DispatchQueue.global().async{
    print("Hello")
    group.leave()
}

```

```
}  
group.wait()  
print("World")
```

В начале выполнения мы инициализируем объект `DispatchGroup` вызвав его конструктор. Внутренний счетчик после инициализации равен нулю.

```
Group counter == 0
```

Увеличиваем его вызовом функции `enter`.

```
Group counter == 1
```

Далее получаем фоновую очередь и ставим на нее задачу. У нас создается фоновый поток, который будет выполнять эту задачу. А первый поток вызывает функцию `wait` у группы и блокируется до тех пор, пока счетчик не обнулится.

```
Thread1  
let group = DispatchGroup()  
group.enter()  
DispatchQueue.global().async{  
    print("Hello")  
    group.leave()  
}  
group.wait()  
print("World")  
  
Thread2  
{  
    print("Hello")  
    group.leave()  
}
```

В это время на втором потоке вызывается `print Hello`.

А затем у группы вызывается `leave`, который уменьшает счетчик на 1, что разблокирует первый поток.

```
Group counter == 0
```

И теперь уже выполняется `print World`.

`DispatchGroup` также предоставляет метод `notify`. Он принимает два аргумента - очередь и замыкание.

```
let group = DispatchGroup()  
group.enter()  
DispatchQueue.global().async {  
    print("Hello")  
    group.leave()  
}  
group.notify(queue: .main) {  
    print("All tasks are done")  
}
```

Когда все задачи в `DispatchGroup` завершатся, это замыкание будет вызвано в контексте той очереди, которую мы указываем первым параметром. Полезно, когда нет необходимости или возможности

использовать синхронное блокирующее ожидание через wait.  
Как делать не надо:

```
// на главной очереди
DispatchQueue.main.sync{ //deadlock
    // ...
}

// на главной очереди
DispatchQueue.global().sync{ // блокирующее ожидание
    // ...
}
```

Плохая идея вызывать sync при выполнении на главном потоке. Если мы выполняем код на главном потоке вызываем sync у главной очереди, то тут же попадем в deadlock и приложение дальше работать уже не будет. А если мы вызываем sync у какой-то другой очереди, то оно, конечно, работать будет, но поскольку sync блокирует выполнение потока, ваше приложение не сможет отвечать на действия пользователя в течение времени ожидания.

Но часто все же нужно сделать что-то синхронно на главном потоке при получении данных откуда-нибудь. В этом случае, конечно, нужно использовать метод sync на главной очереди. Но бывают ситуации, когда мы не можем быть уверены, на каком потоке код будет выполняться, а выполнить кусочек кода синхронно на главном потоке нужно. В этом случае вам может помочь такая простая вспомогательная функция. Она проверяет: если мы прямо сейчас выполняемся на главном потоке, то запустит замыкание, просто вызвав его, а иначе попросит главную очередь сделать это. Такая функция гарантированно защищает вас от дедлока.

```
func syncOnMainThread<T>(execute block: () throws -> T) rethrows -> T {
    if Thread.isMainThread {
        return try block()
    }
    return try DispatchQueue.main.sync(execute: block)
}
```

А иногда бывает полезным отложить какое-нибудь действие на какое-то время. Для этого можно воспользоваться методом asyncAfter у той очереди, на которой мы хотим выполнить это действие.

```
let queue = DispatchQueue.main
queue.asyncAfter(deadline: .now() + 5.5) {
    print("Hello world")
}
```

Обратите внимание, как записывается аргумент deadline. Тип этого аргумента - DispatchTime, который можно инициализировать вызовом статической функции now, а затем сложить с числом, которое будет проинтерпретировано как число секунд. Таким образом, такая запись означает: "Через 5 с половиной секунд на главной очереди вызвать функцию, которая напечатает hello world".

А если Вам необходимо выполнить на очереди какую-то задачу строго после всех предыдущих задач этой очереди, то можно воспользоваться специальным флагом barrier, который можно указать при вызове функции async.

```
let queue = DispatchQueue.global(qos: .utility)
queue.async{
    // task1
}
```

```
queue.async{
    // task2
}
queue.async(flags: .barrier) {
    print("All tasks are done.")
}
```

В этом примере мы не можем сказать, какая задача выполнится раньше, а какая позже. Но мы точно знаем, что третья задача начнется не раньше, чем они закончатся.

Вот, пожалуй, и все, что нужно знать о Grand Central Dispatch. Это богатый и достаточно удобный API для управления многопоточностью. А в следующей части мы посмотрим на Операции и сравним их с GCD.

### 1.3. Операции

Последней темой, которую мы затронем в этой лекции, будет еще одна высокоуровневая технология управления многопоточностью - операции и OperationQueue.

Она похожа на Grand Central Dispatch и разделяет многие ее концепции. Даже правильно будет сказать, что это GCD похожа на Операции, ведь они появились в 2007, за два года до GCD. Операции уже были в версии iPhone OS 2, которая была первой версией iOS, поддерживающей приложения, написанные сторонними разработчиками. И несмотря на то, что Операции появилась раньше, эта технология предоставляет некоторые механизмы, которых в GCD может не хватать. Но обо всем по порядку.

Первое, что нам нужно определить - сам термин **Операция**. Операция представляет собой законченную задачу. Она является абстрактным классом, который предоставляет вам удобную, потокобезопасную структуру для моделирования состояния операции, ее приоритета и зависимостей от других Операций.

У операции есть свой жизненный цикл.

Он начинается с того, что она создается. Но созданная операция никак не взаимодействует с окружающим миром и не выполняется. Для того, чтобы запустить операцию, ее нужно добавить в какую-то очередь OperationQueue. После этого операция будет дожидаться, когда все ее зависимости будут выполнены. Как только это происходит, операция переходит в состояние Ready. Здесь она будет дожидаться, когда очередь освободится и возьмет эту операцию на выполнение, а затем она переходит в состояние Executing. А когда выполнение завершится, состояние сменится на Finished. Кроме того, из каждого состояния операция может быть отменена, и в таком случае она становится Cancelled.

У операции могут быть зависимости в виде других операций.

Например, операция GetDataOperation зависит от операции FetchCacheOperation, которая, в свою очередь, зависит от AcquireSettingsOperation.

Зависимости добавляются с помощью вызова метода addDependency у операции, которой эту зависимость нужно добавить.

```
let getDataOperation = GetDataOperation()
let fetchCacheOperation = FetchCacheOperation()
```

```
let acquireSettingsOperation = AcquireSettingsOperation()

getDataOperation.addDependency(fetchCacheOperation)
fetchCacheOperation.addDependency(acquireSettingsOperation)
```

Очередь `OperationQueue` концептуально похожа на `DispatchQueue` из GCD, но у нее есть некоторые преимущества. Важной особенностью `OperationQueue` является возможность установить параметр `maxConcurrentOperationCount`, ограничивающий количество операций, которые могут выполняться "параллельно". `QualityOfService`, или класс обслуживания, который мы рассматривали ранее, так же может быть применен к `OperationQueue`. Кроме того, можно управлять параметром `isSuspended`, установка которого в значение `true` приостанавливает выполнение операций на данной очереди.

```
let queue = OperationQueue()
queue.maxConcurrentOperationCount = 1
queue.qualityOfService = .background
queue.isSuspended = true
```

Основной способ взаимодействия с очередью - это добавление операций в нее. Происходит это с помощью вызова метода `addOperation`.

```
queue.addOperation(getDataOperation)
```

Параметр `maxConcurrentOperationCount` очень важен, и вот почему.

Если мы установим его, например, в значение 5, то на этой очереди сможет выполняться "параллельно" 5 операций. Кажется, не сложно. Если же ничего не устанавливать в это свойство, то оно будет иметь значение `Default`, которое позволит выполнять столько операций одновременно, сколько это позволяет iOS.

Но следует помнить, что этот параметр не заставляет очередь действительно запускать столько операций, сколько мы здесь укажем. Это ограничение сверху. Мы, конечно, можем указать здесь 100, 1000 или 1000000, но параллельно будут выполняться только несколько операций.

Если Вам действительно нужно завести очень много потоков, то придется пользоваться какими-нибудь более низкоуровневыми инструментами, такими как `pthread` или класс `Thread`. Но делать этого не стоит, потому что большое число потоков приводит к тому, что планировщик операционной системы начинает тратить очень много времени на сами переключения между потоками, и даже ухудшает общую производительность.

А вот установка параметра `maxConcurrentOperationCount` в единицу делает эту очередь последовательной, то есть все операции на ней будут выполняться одна за одной.

Примером последовательной очереди является очередь `OperationQueue main`, ассоциированная с главным потоком приложения.

```
let setLabelTextOperation = BlockOperation {
    label.text = "Hello world"
}
OperationQueue.main.addOperation(setLabelTextOperation)
```

Все операции, взаимодействующие с UI, должны выполняться на ней.

Операции бывают синхронные и асинхронные. Для создания синхронной операции достаточно определить класс - наследник `Operation`, в котором переопределить метод `main`.

```
class MySynchronousOperation: Operation {
    override func main() {
        print("Hello World")
    }
}

let mySynchronousOperation = MySynchronousOperation()
queue.addOperation(mySynchronousOperation)
```


Синхронные операции запускают этот метод на потоке, предоставленном очереди, и после того, как выполнение будет завершено, сразу переходят в состояние Finished.

А вот асинхронные операции выполняются по-другому. Сразу после старта операции они возвращают управление потоку, но сама операция не становится завершенной.

```
class AsyncOperation: Operation {
    override var isAsynchronous: Bool {
        return true
    }
    override var isFinished: Bool
    override var isExecuting: Bool
    override func start() {
        // ...
    }
}
```

Завершение происходит, когда свойство isFinished объекта Operation становится истинным. А добиться этого не так просто. Дело в том, что поля isFinished, isExecuting и другие - read-only свойства, так что нам нужно будет переопределить и функцию запуска, и свойства, определяющие текущее состояние операции. И надо не забыть переопределить свойство isAsynchronous так, чтобы оно возвращало Истину.

Удобно, если логика, реализующая асинхронную операцию, будет выделена в подкласс - AsyncOperation. Мы не будем разбирать полную его реализацию, но он будет доступен в материалах к этой лекции. Может показаться, что GCD и Операции очень похожи, но все же у них есть важные отличия, которые стоит еще раз отметить.



pres7.png

В GCD мы не можем указывать или ограничивать число потоков, ассоциированных с очередью. Либо один при использовании Serial очереди, либо несколько, но заранее неизвестно, сколько, если очередь Concurrent. Нельзя попросить у GCD очереди выполнять, например, ровно 17 или не более 17 операций одновременно. А в OperationQueue можно указать лимит, для этого есть параметр maxConcurrentOperationCount.

Что касается отмены - Операции ее поддерживают. Причем для синхронных операций отмена работает прямо из коробки, а для асинхронных может быть легко добавлена. В GCD тоже есть возможность отменять задачи, но использовать эту возможность не слишком удобно, поэтому и пользуются этим редко. Дело в том, что отменить задачу в GCD можно, только если эта задача была создана не просто блоком или функцией, а через создание экземпляра класса DispatchWorkItem. Этот класс предоставляет функцию cancel, аналогично тому, что мы видели в операциях. Если задача еще не

начала выполняться, то, вызвав эту функцию, задачу можно отменить и она будет снята с очереди. Другим важным отличием операций является механизм зависимостей, которого нет в GCD. Вместе это делает работу с GCD немного проще, если мы точно знаем, что нам не понадобится что-нибудь, что доступно только в Операциях.

На этом мы завершаем тему многопоточности. Конечно, это не все, о чем можно было бы рассказать, но это необходимый минимум, который нужен для того, чтобы ориентироваться в средствах управления многопоточностью при разработке под iOS. А на следующей неделе Вы научитесь работать с сетью, отправлять запросы в интернет и работать с HTTP API.