Параллельное программирование. GCD

Изучаем библиотеку Grand Central Dispatch и упрощаем управление потоками

[Введение](#_e27dxsqemq0)

[Очереди](#_v5qzwkfgqqd)

[Глобальные](#_607ka93fnuxy)

[Последовательные и параллельные](#_go9wx0lpeu6b)

[Синхронное и асинхронное выполнение заданий](#_sv9rszv5jxrs)

[Создание своих очередей](#_f5cjtgmoeip7)

[Проблемы синхронизации](#_eqlcb6c618iw)

[Перенос сложной UI задачи из главного потока в фоновый](#_1kprr2kyt0d4)

[Ожидание выполнения всех параллельных задач](#_qqtn13yadn8r)

[Гонка состояний](#_9ike7vh6m7r4)

[Домашнее задание](#_s1tptljd9vb)

[Дополнительные материалы](#_xh9je3heix2t)

[Используемая литература](#_uvp6qax5r1ok)

# Введение

На прошлом уроке мы познакомились с новыми понятиями: потоками, процессами, петлей событий, асинхронным и многопоточным кодом и другими. Этого достаточно для успешного разделения кода на потоки, но работать с классом **Thread** не совсем удобно. Приходится держать в голове, сколько потоков необходимо создать, как разделять код между ними, добавлять задания в поток и синхронизировать данные, отслеживать завершение потока. Стандартных решений для этих задач нет.

Чтобы упростить написание конкретного кода, корпорация Apple создала библиотеку Grand Central Dispatch. Она полностью инкапсулирует в себе работу с потоками, взамен предоставляя разработчику интерфейс очередей и заданий.

При работе с GCD вы оперируете заданиями блока кода, которые помещаются в очереди. Они могут выполняться параллельно или последовательно. Очереди работают с разной скоростью. Задания, помещенные в главную очередь, всегда выполняются в главном потоке. Другие распределяются по потокам на усмотрение библиотеки.

Самый простой способ выполнить блок кода в другом потоке выглядит так.

|  |
| --- |
| DispatchQueue.global().async {  for \_ in (0..<10) {  print("😈")  } } |

Так получаем ссылку на глобальную очередь, вызываем метод “async” и передаем ему замыкание, которое и будет выполнено в рамках этой очереди. На первый взгляд очень мало отличий от Thread, но в дальнейшем мы увидим, что это не так.

Важно запомнить, что ***очередь не равна потоку***. Задания из одной очереди могут выполняться на разных потоках, а задачи из разных очередей - на одном потоке.

# Очереди

## Глобальные очереди

Чтобы выполнять задания, нужна очередь. Ее можно создать или получить одну из 6 стандартных очередей, которые существуют по умолчанию в каждом ios-приложении: одна из них уникальная, связанная с главным потоком, и 5 дополнительных с разными приоритетами выполнения.

1. **DispatchQueue.main** - последовательная очередь Main queue, в которой происходят все операции с пользовательским интерфейсом (UI). Поэтому функции и замыкания, направленные на UI, помещаем на нее. **Main queue** имеет наивысший приоритет среди глобальных очередей.
2. **DispatchQueue.global(qos: .userInteractive)** - очередь для заданий, которые отвечают за взаимодействие с пользователем в данный момент и занимают мало времени - например, для анимации. Представим: пользователь водит пальцем по экрану, а приложению необходимо просчитать что-то, связанное с интенсивной обработкой изображения. И расчет размещается в этой очереди. Тогда результат немного отстает от положения пальца на экране, так как расчеты требуют времени. Но по крайней мере, **Main queue** реагирует на прикосновения. Эта очередь имеет очень высокий приоритет, но ниже, чем у **Main queue**.
3. **DispatchQueue.global(qos: .userInitiated)** - очередь для заданий вне интерактивного события, которые инициируются пользователем и требуют обратной связи для дальнейшего взаимодействия. Выполнение может занять несколько секунд. Эта очередь имеет высокий приоритет, но ниже, чем у предыдущей очереди.
4. **DispatchQueue.global(qos: .utility)** - для заданий, которым нужно время для выполнения и не требуется немедленная обратная связь: например, для загрузки или очистки базы данных. Иначе говоря, очередь занимается тем, о чем пользователь не просит, но оно необходимо приложению. Задание может занять от несколько секунд до нескольких минут. Приоритет ниже, чем у предыдущей очереди.
5. **DispatchQueue.global(qos: .background)** - очередь для фоновых задач, не связанных с визуализацией и не критичных по времени исполнения: например, backups или синхронизации с web-сервисом. Выполняется от считанных минут до нескольких часов и только тогда, когда не требуется иное обслуживание. Приоритет самый низкий среди всех глобальных очередей.
6. **DispatchQueue.global(qos: .default)** - очередь по умолчанию, ее приоритет автоматически выбирается между **userInitiated** и **utility**. Ее также можно получить, не передавая параметры в метод **global** - **DispatchQueue.global().**

Выполним несколько заданий на этих очередях.

|  |
| --- |
| DispatchQueue.global(qos: .userInteractive).async {  for \_ in (0..<10) {  print("😇")  } }  DispatchQueue.main.async {  for \_ in (0..<10) {  print("😈")  } } |

Как мы видим, задачи выполняются почти одновременно. Ресурсов хватает.

|  |
| --- |
| 😇 😈 😇 😈 😈 😇 😈 😇 😈 😇 😈 😇 😈 😇 😈 😇 😈 😇 😈 😇 |

Теперь запустим 4 задания на 4 очередях одновременно.

|  |
| --- |
| DispatchQueue.global(qos: .userInteractive).async {  for \_ in (0..<10) {  print("😇")  } }  DispatchQueue.main.async {  for \_ in (0..<10) {  print("😈")  } }  DispatchQueue.global(qos: .userInitiated).async {  for \_ in (0..<10) {  print("👻")  } }  DispatchQueue.global(qos: .utility).async {  for \_ in (0..<10) {  print("👽")  } } |

В результате ангелы и демоны выводятся так же парами, призрак появляется уже реже - на двух демонов и двух ангелов будет всего один призрак. Больше всего не повезло инопланетянам: они появились всего два раза вместе с остальными, а большинство пришельцев вышли в свет уже после того, как завершились все остальные задания. В этом примере приоритет очередей виден гораздо лучше.

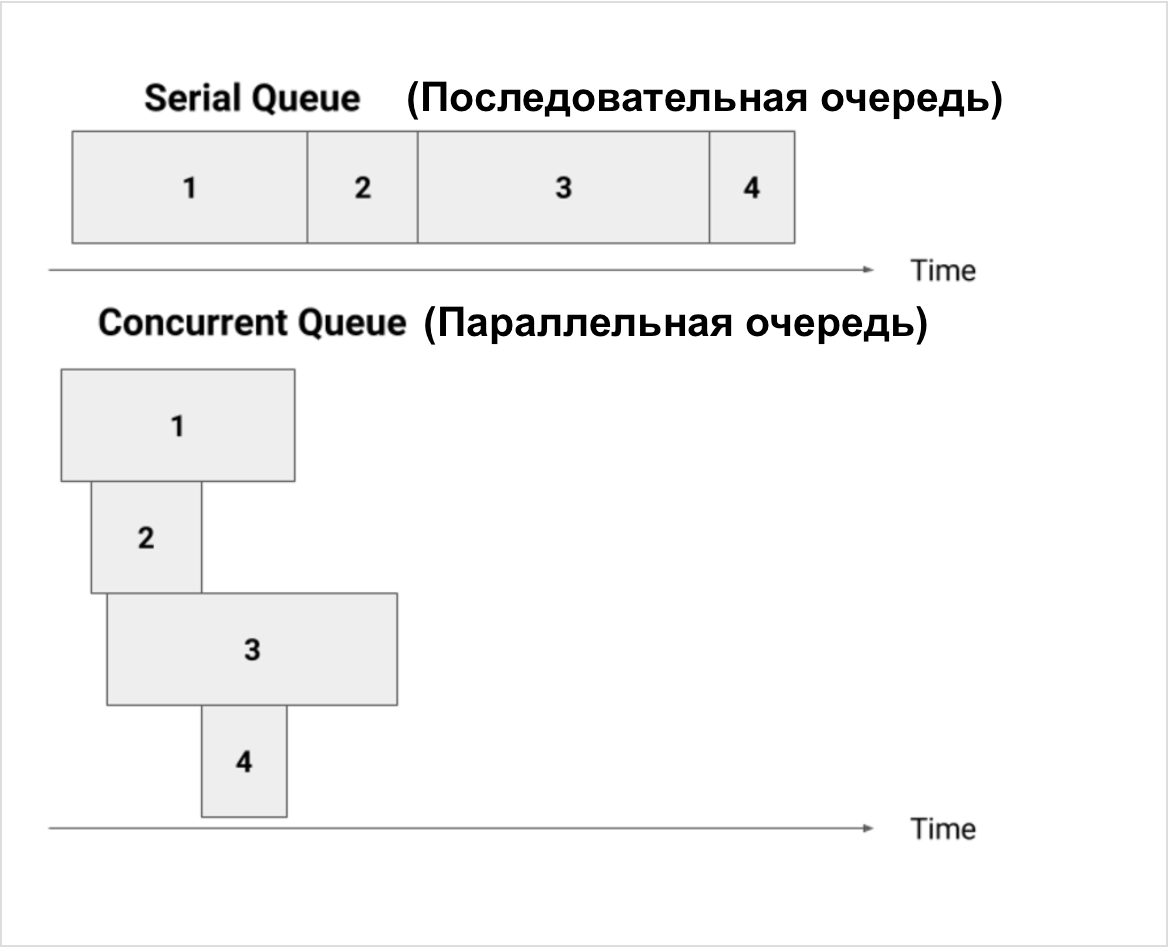
|  |
| --- |
| 😇 😈 👻 👽 😈 😇 👻 👽 😈 😇 😈 😇 👻 😈 😇 😈 👻 😇 😈 😇 😈 👻 😇 😈 👻 😇 😈 👻 😇 👻 👻 👻 👽 👽 👽 👽 👽 👽 👽 👽 |

Важно понимать, что бесполезно считать, сколько призраков было выведено на одного демона или ангела - это соотношение непостоянно. Порядок чередования задач зависит от множества факторов: количества потоков в приложении, их загруженности, наличия свободных ресурсов процессора и других. Главное - знать, что задания в одних очередях будут выполняться быстрее, чем в других.

## Последовательные и параллельные

Последовательные очереди (**serial queues**) “вытягивают” задание (замыкание), которое находится на вершине очереди, и дают ему работать до завершения, а затем приступают к следующему.

Параллельные очереди (**concurrent queues**) тоже берут в работу задание с вершины, запуская его в определенном потоке. Если у системы есть еще ресурсы, то она сразу приступает к следующему заданию из очереди, посылая его в другой поток, в то время как первая функция еще работает. Так система может “вытянуть” целый ряд задач.



Мы видим, что на **serial** - последовательной очереди - завершение заданий происходит строго в том порядке, в каком они поступали на выполнение, в то время как на **concurrent** - параллельной очереди - задания заканчиваются непредсказуемым образом. Кроме того, общее время выполнения группы заданий на **serial**-очереди значительно больше, чем у аналогичной на **concurrent**-очереди. Ведь при последовательном принципе в каждый конкретный момент времени выполняется только одно задание, а при параллельном число задач, решаемых одновременно, может меняться.

Рассмотрим два примера. В первом добавим в главную очередь 5 заданий, каждое из которых будет выводить одну букву слова “Hello”.

|  |
| --- |
| DispatchQueue.main.async { print("H") } DispatchQueue.main.async { print("e") } DispatchQueue.main.async { print("l") } DispatchQueue.main.async { print("l") } DispatchQueue.main.async { print("o") } |

Так как в главной очереди задания выполняются в порядке добавления, то в консоли мы увидим точно такую же последовательность символов - слово “hello”.

|  |
| --- |
| H e l l o |

Во втором примере сделаем то же самое, только задания будем добавлять в глобальную очередь.

|  |
| --- |
| DispatchQueue.global().async { print("H") } DispatchQueue.global().async { print("e") } DispatchQueue.global().async { print("l") } DispatchQueue.global().async { print("l") } DispatchQueue.global().async { print("o") } |

Так как очередь параллельная, все задания начнут выполняться одновременно, а завершаться первыми будут наиболее быстрые. Хотя в нашем примере все задания имеют одинаковое время выполнения, порядок вывода букв будет каждый раз разный.

|  |
| --- |
| e H o l l |

Впрочем, при доле везения задания могут выполниться и в порядке добавления.

Теперь увеличим время на выполнение заданий с буквами “e” и “o” - добавим в них “sleep”.

|  |
| --- |
| DispatchQueue.global().async { print("H") } DispatchQueue.global().async {  sleep(1)  print("e") } DispatchQueue.global().async { print("l") } DispatchQueue.global().async { print("l") } DispatchQueue.global().async {  sleep(1)  print("o") } |

В таком варианте они всегда будут выводиться в консоль последними, так как выполняются дольше.

|  |
| --- |
| H l l e o |

Это демонстрация закономерностей работы очередей. На практике нет необходимости вручную упорядочивать вывод параллельной очереди.

## Синхронное и асинхронное выполнение заданий

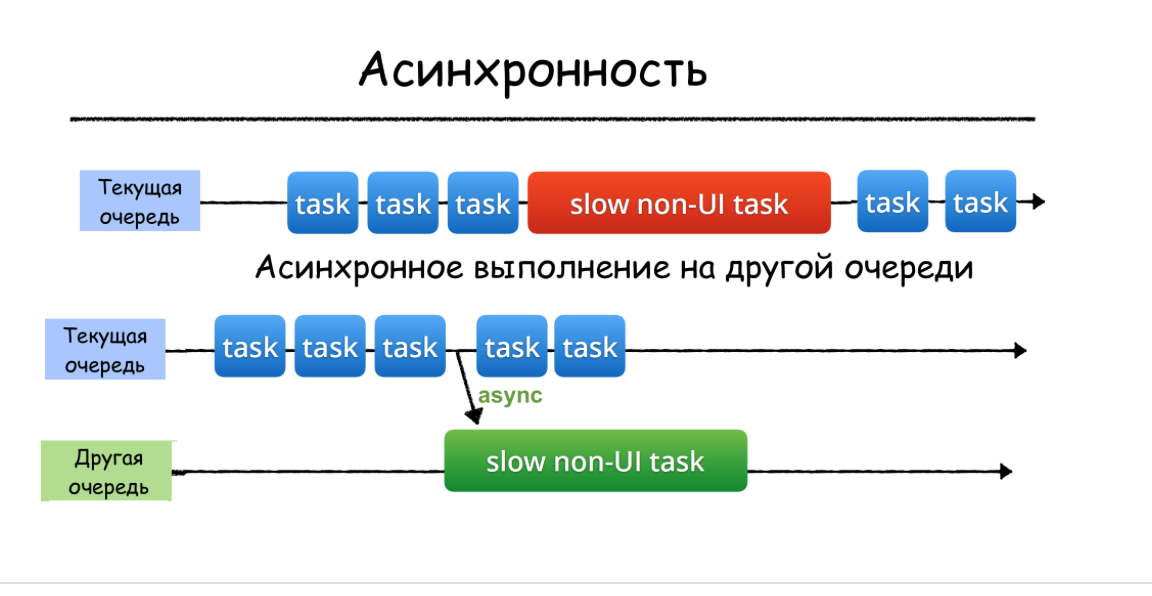
Каждый раз, когда вы ставите задание в очередь, вы сами находитесь в какой-то из очередей. По умолчанию это главная, но может быть и любая другая. С помощью методов “sync” и “async” можно управлять выполнением текущей очереди относительно установленного задания.

Находясь на главной очереди, асинхронно ставим задачу в фоновую и после выводим в консоль ангела.

|  |
| --- |
| DispatchQueue.global().async {  print("😈") }   print("😇") |

|  |
| --- |
| 😇 😈 |

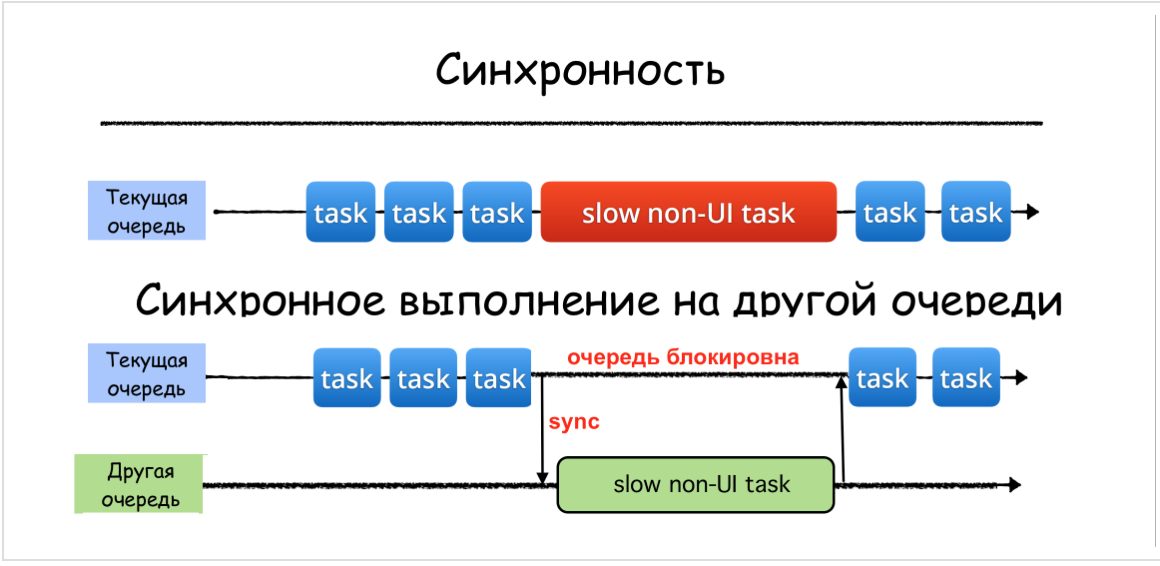
Сначала выходит ангел, потом демон. Это случилось потому, что главная очередь продолжила свою работу и успела выполнить следующую инструкцию прежде, чем фоновая. Вывод: главная очередь после постановки задачи продолжила жить своей жизнью.



Повторим пример, но задание установим уже синхронно.

|  |
| --- |
| DispatchQueue.global().sync {  print("😈")  }   print("😇") |

|  |
| --- |
| **😈 😇** |

Ангел и демон поменялись местами. Метод “sync” поставил главную очередь на паузу до полного выполнения задания. 

Использовать метод “sync” надо, только если необходимо дождаться завершения задачи.

## Создание своих очередей

Кроме 5 стандартных очередей, можно создавать собственные таким образом:

|  |
| --- |
| let myQueue = DispatchQueue(label: "myQueue")  myQueue.async {  for \_ in (0..<10) {  print("😈")  }  } |

При этом надо указать текстовое название очереди. Она будет последовательной (**serial**) и обладать приоритетом, равным глобальной очереди (**userInitiated** или **utility**).

**Создавать свои очереди имеет смысл только по двум причинам:**

1. Нужна фоновая последовательная очередь.
2. Много многопоточного кода, который хочется логически организовать.

Созданные разработчиком очереди не являются самостоятельными. В итоге они все равно отправляют свои задачи в одну из глобальных очередей.

В примере выше очередь создана с использованием минимума параметров. Теперь воспользуемся всеми.

|  |
| --- |
| let myQueue = DispatchQueue(label: "myQueue",  qos: DispatchQoS.utility,  attributes: DispatchQueue.Attributes.concurrent,  autoreleaseFrequency: DispatchQueue.AutoreleaseFrequency.inherit,  target: DispatchQueue.global(qos: DispatchQoS.QoSClass.utility)) |

Получилась параллельная очередь с приоритетом “**utility**”, которая будет отправлять свои задачи в глобальную очередь таким же приоритетом, и ее память будет освобождаться в обычном режиме. Рассмотрим все параметры и их возможные значения:

1. **qos** - приоритет.
2. **attributes** - подразумевает два параметра, можно установить один или сразу оба:
   1. **.concurrent** - преобразует очередь в параллельную. Если его не указать, то очередь будет последовательной;
   2. **.initiallyInactive** - указывает, что созданная очередь будет неактивной. Установленные в нее задания не начнут выполняться, пока вы не активируете очередь.
3. **autoreleaseFrequency** - указывает, как очереди освобождать память:
   1. **.inherit** - память освобождается также, как и в очереди, на которую отправляются задачи;
   2. **.workItem** - память освобождается после выполнения каждого задания;
   3. **.never** - очередь не управляет автоматическим освобождением памяти.
4. **target (целевая очередь)** - очередь, в которую будут отправлены задания.

В большинстве случаев указываются только два параметра: **qos** и **attributes**.

# Проблемы синхронизации

Параллельное выполнение нескольких блоков кода само по себе - довольно простая задача. При этом написание конкурирующего кода считается наиболее сложной в изучении языка и программирования в целом. Дело в том, что выполнить абстрактную задачу в отдельном потоке недостаточно - этот код должен взаимодействовать с данными из других потоков. В этом разделе мы разберем несколько реальных задач, требующих решения.

## Перенос сложной UI задачи из главного потока в фоновый

Одна из самых распространенных задач - вынос расчетов из главного потока в фоновый. Как правило, эти задачи не требуют синхронизации с другими, но их результат должен быть отображен в главном потоке.

На примере отобразим таблицу с изображениями, к каждому из которых применим эффект размытия.

|  |
| --- |
| override func numberOfSections(in tableView: UITableView) -> Int {  return 1 }  override func tableView(\_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {  return 10 }    override func tableView(\_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {  let cell = tableView.dequeueReusableCell(withIdentifier: "SomeCell", for: indexPath)    let inputImage = UIImage(named: "treeSmall")!  let inputCIImage = CIImage(image: inputImage)!    let blurFilter = CIFilter(name: "CIGaussianBlur", withInputParameters: [kCIInputImageKey: inputCIImage])!  let outputImage = blurFilter.outputImage!  let context = CIContext()  let cgiImage = context.createCGImage(outputImage, from: outputImage.extent)    let bluredImage = UIImage(cgImage: cgiImage!)     cell.imageView?.image = bluredImage   return cell } |

Запустим этот пример и увидим, что таблица загружается очень медленно и зависает при прокрутке. Причина в том, что применение фильтров к изображению - очень долгая и ресурсоемкая операция. Это при том, что наше исходное изображение очень маленькое и строк в таблице всего 10. При более объемных исходных приложение вовсе не запустилось бы из-за нехватки ресурсов.

Вынесем обработку изображения в отдельный метод и выполним в отдельном потоке.

|  |
| --- |
| override func tableView(\_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {  let cell = tableView.dequeueReusableCell(withIdentifier: "SomeCell", for: indexPath)    let inputImage = UIImage(named: "treeSmall")!    cell.imageView?.image = inputImage  blur(image: inputImage, andApplyToView: cell.imageView!)   return cell }   func blur(image: UIImage, andApplyToView view: UIImageView) {    DispatchQueue.global(qos: .userInteractive).async {  let inputImage = UIImage(named: "treeSmall")!  let inputCIImage = CIImage(image: inputImage)!    let blurFilter = CIFilter(name: "CIGaussianBlur", withInputParameters: [kCIInputImageKey: inputCIImage])!  let outputImage = blurFilter.outputImage!  let context = CIContext()    let cgiImage = context.createCGImage(outputImage, from: outputImage.extent)    let bluredImage = UIImage(cgImage: cgiImage!)    view.image = bluredImage  } } |

Запускаем пример и видим, что “тормоза” пропали, прокрутка работает отлично. Но размытие не выполняется, сколько ни жди. А вот если нажимать на ячейки, то фильтр к изображениям начнет применяться.

Так произошло потому, что мы заняли изображения в фоновом потоке. Мы как бы тайком, без разрешения пытались подменить исходную картинку на размытую. Нам еще повезло, что в этот момент никто ее не использовал и нас не поймали с поличным - иначе приложение упало бы.

Именно поэтому первое правило многопоточности гласит, что ***изменять UI можно только из главного потока***.

Поставим задание в его очередь.

|  |
| --- |
| func blur(image: UIImage, andApplyToView view: UIImageView) {    DispatchQueue.global(qos: .userInteractive).async {  let inputImage = UIImage(named: "treeSmall")!  let inputCIImage = CIImage(image: inputImage)!    let blurFilter = CIFilter(name: "CIGaussianBlur", withInputParameters: [kCIInputImageKey: inputCIImage])!  let outputImage = blurFilter.outputImage!  let context = CIContext()    let cgiImage = context.createCGImage(outputImage, from: outputImage.extent)    let bluredImage = UIImage(cgImage: cgiImage!)    DispatchQueue.main.async {  view.image = bluredImage  }  }  } |

Таким образом мы решили проблему возвращения в главный поток после асинхронной операции.

## Ожидание выполнения всех параллельных задач

При выполнении нескольких параллельных задач может возникнуть необходимость выполнить какой-либо код после их завершения.

В предыдущем примере мы избавились от “фризов”, но изображения стали размываться заново при прокрутке.

Создадим массив, где будем хранить изображения при запуске приложения. Зададим параллельное размытие всех картинок, а по завершении обновим таблицу.

|  |
| --- |
| var images = [  UIImage(named: "treeSmall")!,  UIImage(named: "treeSmall")!,  UIImage(named: "treeSmall")!,  UIImage(named: "treeSmall")!,  UIImage(named: "treeSmall")!,  UIImage(named: "treeSmall")!,  UIImage(named: "treeSmall")!,  UIImage(named: "treeSmall")!,  UIImage(named: "treeSmall")!,  UIImage(named: "treeSmall")!  ] |

|  |
| --- |
| override func numberOfSections(in tableView: UITableView) -> Int {  return 1 }  override func tableView(\_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {  return 10 }    override func tableView(\_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {  let cell = tableView.dequeueReusableCell(withIdentifier: "SomeCell", for: indexPath)    cell.imageView?.image = images[indexPath.row]   return cell } |

|  |
| --- |
| func blurImages() {  var bluredImages = images  let dispatchGroup = DispatchGroup()    for element in bluredImages.enumerated() {    DispatchQueue.global().async(group: dispatchGroup) {    let inputImage = element.element  let inputCIImage = CIImage(image: inputImage)!    let blurFilter = CIFilter(name: "CIGaussianBlur", withInputParameters: [kCIInputImageKey: inputCIImage])!  let outputImage = blurFilter.outputImage!  let context = CIContext()    let cgiImage = context.createCGImage(outputImage, from: outputImage.extent)    bluredImages[element.offset] = UIImage(cgImage: cgiImage!)  }  }    dispatchGroup.notify(queue: DispatchQueue.main) {  self.images = bluredImages  self.tableView.reloadData()  }   } |

Копируем массив - получаем временный, в нем для каждого изображения создаем задачу размытия и добавляем все задания в параллельную очередь. При этом таблица будет работать с исходными изображениями в изначальном массиве. Для контроля за окончанием всех заданий создаем “**DispatchGroup**” - специальный объект, который отслеживает связанные с ним задачи. Добавляя задания в очередь, указываем группу.

И в конце вызываем метод “**notify**” у группы диспетчеризации. В нем указываем очередь, на которой надо выполнить замыкание - главную. В самом замыкании просто переносим все размытые изображения из временного массива в основной и обновляем таблицу. Останется только вызвать этот метод при загрузке контроллера.

|  |
| --- |
| override func viewDidLoad() {  super.viewDidLoad()    blurImages() } |

## Гонка состояний

Это одна из фундаментальных проблем конкурирующего кода. Ее суть в том, что операция может требовать определенного состояния системы, при этом в промежутке между проверкой условий и выполнением операции состояние способно измениться.

Рассмотрим два примера: один из сферы IT, второй из повседневной жизни.

Представим работу с интернет-магазином: при покупке товара сначала проверяется состояние внутреннего кошелька покупателя. Если денег достаточно, заказ помечается как оплаченный. А что, если пользователь одновременно оформит заказ с нескольких компьютеров? При обработке заказов так же одновременно состоится несколько проверок на наличие средств - и все они завершатся успехом. Только на следующем шаге статус заказа будет подтвержден и спишутся средства. Это, кстати, реальный фактор уязвимости некоторых магазинов.

Или пример, знакомый водителям: едем на машине и начинаем перестраиваться в соседний ряд. Смотрим в зеркало - помех нет. Возвращаем взгляд на дорогу спокойно начинаем маневр. Но в промежутке между этими двумя событиями на соседней полосе не в меру резвый мотоциклист может поравняться с нами - вот и аварийная ситуация.

Это все типичные ошибки гонки состояний, которым подвержены многие системы. Они могут проявляться как часто, так и очень редко, и обнаружить их не всегда просто.

Для демонстрации данной ошибки и ее решения напишем простой пример. Создадим хранилище документов, присваивая каждому номер в соответствии с порядком добавления в хранилище.

Начнем со структуры документа.

|  |
| --- |
| struct Document: CustomStringConvertible {  let id: Int  let name: String    var description: String {  return "\(id) - \(name)"  } } |

Затем пишем небольшой цикл, перебирающий алфавит: буквы станут именами документов. На каждой итерации получаем id предыдущего документа, увеличиваем его на 1 и делаем id нового документа. В конце выводим в консоль получившийся массив.

|  |
| --- |
| let firstChar = UnicodeScalar("А").value let lastChar = UnicodeScalar("Я").value  var documents = [Document]() for charCode in firstChar...lastChar {  let docName = String(UnicodeScalar(charCode)!)  let lastId = documents.last?.id ?? 0  let newId = lastId + 1  let doc = Document(id: newId, name: docName)  documents.append(doc) } print(documents) |

|  |
| --- |
| [1 - А, 2 - Б, 3 - В, 4 - Г, 5 - Д, 6 - Е, 7 - Ж, 8 - З, 9 - И, 10 - Й, 11 - К, 12 - Л, 13 - М, 14 - Н, 15 - О, 16 - П, 17 - Р, 18 - С, 19 - Т, 20 - У, 21 - Ф, 22 - Х, 23 - Ц, 24 - Ч, 25 - Ш, 26 - Щ, 27 - Ъ, 28 - Ы, 29 - Ь, 30 - Э, 31 - Ю, 32 - Я] |

Как мы можем увидеть, все id расположены по порядку и соответствуют очередности добавления документов в хранилище.

Изменим пример так, чтобы итерации были отдельными заданиями и исполнялись одновременно. Добавим **DispatchGroup**, чтобы отследить окончание работы и вывести результат на экран.

|  |
| --- |
| let firstChar = UnicodeScalar("А").value let lastChar = UnicodeScalar("Я").value let dispatchGroup = DispatchGroup()   for charCode in firstChar...lastChar {  DispatchQueue.global().async(group: dispatchGroup) {  let docName = String(UnicodeScalar(charCode)!)  let lastId = self.documents.last?.id ?? 0  let newId = lastId + 1  let doc = Document(id: newId, name: docName)  self.documents.append(doc)  }  }   dispatchGroup.notify(queue: DispatchQueue.main) {  print(self.documents) } |

В результате получаем на выходе нечто непрезентабельное.

|  |
| --- |
| [1 - Н, 1 - Е] |

Всего две буквы, причем абсолютно случайные, да еще и с одинаковыми номерами. А может выйти и ошибка выполнения программы.

На такой результат повлияло то, что все задачи запустились параллельно. А значит, одновременно считали текущий номер последнего элемента, который в пустом массиве был равен 0. Потом дружно увеличили его на 1 и все вместе попытались записать данные в массив.

Если программа выдала несколько букв и цифр, значит операции все же разошлись немного во времени и просто перезаписали друг друга в памяти массива. Если же вышла ошибка - то операции были абсолютно одновременны и пытались писать одни и те же байты в памяти компьютера.

Все это следствие того, что массив - не потокобезопасная структура данных. Строго говоря, таковы все переменные, свойства и структуры в swift. Одновременное изменение их значений из нескольких потоков может привести к ошибке. У примитивных переменных этот шанс намного ниже, но на везение в реальной программе полагаться нельзя.

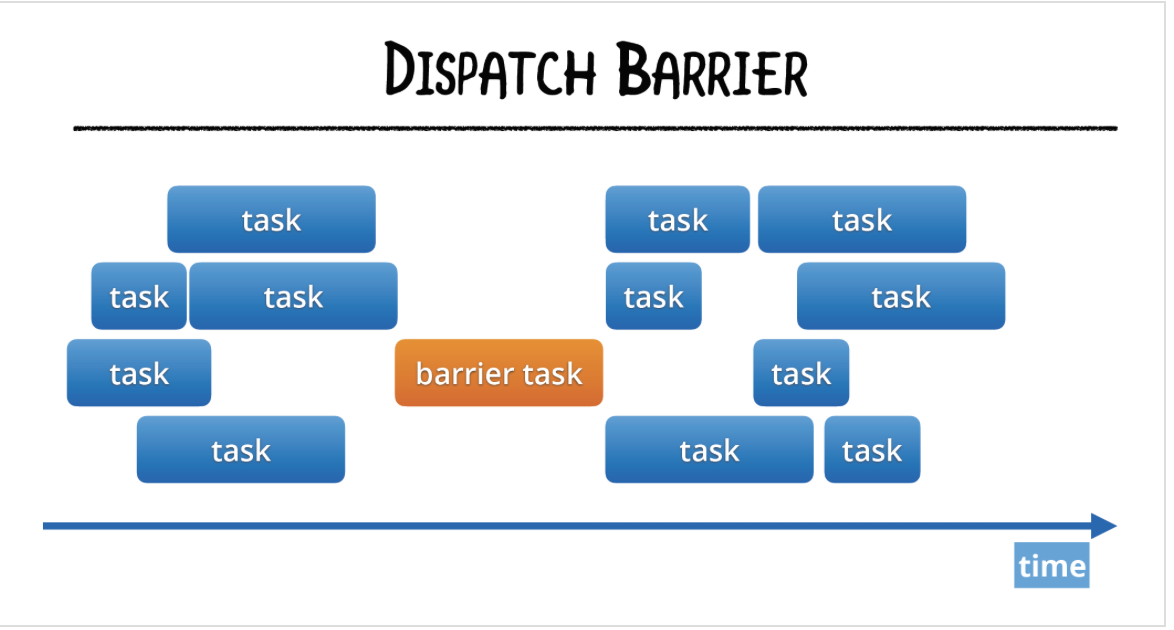
Напишем хранилище для документов и сделаем его потокобезопасным.

|  |
| --- |
| class DocumentStore: CustomStringConvertible {  private var documents = [Document]()  private let syncQueue = DispatchQueue(label: "DocumentStoreSyncQueue", attributes: .concurrent)    func getDocument(byId id: Int) -> Document? {  var document: Document?  syncQueue.sync {  if let index = self.documents.index(where: { $0.id == id }) {  document = self.documents[index]  }  }  return document  }    func getLastDocument() -> Document? {  var document: Document?  syncQueue.sync {  document = self.documents.last  }  return document  }    func append(document: Document) {  syncQueue.async(flags: .barrier) {  self.documents.append(document)  }  }    var description: String {  var description = ""  syncQueue.sync {  description = self.documents.reduce("") { $0 + $1.description + ", " }  }  return description  } } |

Разберем код хранилища. Объявлены два приватных свойства: **documents** - массив для хранения документов; **syncQueue** - очередь, на которой будут проходить все операции с массивом, чтение и запись. Важно, что больше никаких операции на этой очереди не будет и доступ к ней имеется только внутри хранилища. Еще имеет значение, что очередь параллельная.

Далее описываем методы для доступа к элементам массива: **getDocument** и **getLastDocument**. Это обычные методы поиска элемента в массиве, но выполняются синхронно они на очереди **syncQueue**. Значит для того, кто будет вызывать методы получения данных, будет незаметно выполнение в другом потоке.

Особые манипуляции мы совершаем в методе добавления элемента в хранилище - “**append**”. В нем ставим задачу добавления элемента в конец массива, в очередь **syncQueue**. Но эта задача выполняется асинхронно, так как может занять длительное время, а ждать ее завершения нам ни к чему. Самое важное - указываем флаг при добавлении задачи - в данном примере **.barrier**. Благодаря ему задачи выполняются в одиночку даже на параллельных очередях. Они дожидаются завершения предшествующих, а потом выполняются сами, и очередь начинает работать в штатном режиме.



В итоге, мы сможем читать данные из массива одновременно из любого количества потоков - сама по себе операция чтения не может навредить. Но на время записи все другие операции будут остановлены: ни запись из другого потока, ни чтение не смогут быть выполнены.

Переведем наш код работы с документами на потокобезопасное хранилище.

|  |
| --- |
| let firstChar = UnicodeScalar("А").value let lastChar = UnicodeScalar("Я").value let dispatchGroup = DispatchGroup()  let documentStore = DocumentStore()   for charCode in firstChar...lastChar {  DispatchQueue.global().async(group: dispatchGroup) {  let docName = String(UnicodeScalar(charCode)!)  let lastId = self.documentStore.getLastDocument()?.id ?? 0  let newId = lastId + 1  let doc = Document(id: newId, name: docName)  self.documentStore.append(document: doc)  }  }   dispatchGroup.notify(queue: DispatchQueue.main) {  print(self.documentStore) } |

В результате мы получим работающий код. Но, к сожалению, работающий неверно.

|  |
| --- |
| 1 - Е, 1 - Р, 1 - Ю, 1 - Я, 1 - Ъ, 1 - И, 1 - Ж, 1 - Щ, 1 - Н, 1 - П, 1 - Г, 1 - Й, 1 - З, 1 - М, 1 - Л, 1 - Ш, 1 - О, 1 - Б, 1 - Д, 1 - К, 1 - У, 1 - Т, 1 - В, 1 - Х, 1 - Ф, 1 - Ц, 1 - Ч, 1 - С, 1 - А, 1 - Ы, 1 - Ь, 1 - Э, |

В хранилище размещены все элементы, программа не выдает ошибок, но номер у всех документов одинаковый. Так получилось, потому что каждая задача одновременно прочитала номер текущего элемента - 0, а затем одновременно стартовала запись. Записывались они уже по очереди.

Чтобы предотвратить такую ошибку, надо сделать чтение последнего элемента, увеличение на его на единицу и запись элемента ***атомарной операцией***. То есть перенести это все в одну барьерную задачу. Добавим в хранилище соответствующий метод.

|  |
| --- |
| func createDocumenet(fromName name: String) {  syncQueue.async(flags: .barrier) {  let lastId = self.documents.last?.id ?? 0  let newId = lastId + 1  let doc = Document(id: newId, name: name)  self.documents.append(doc)  } } |

И пропишем использование.

|  |
| --- |
| let firstChar = UnicodeScalar("А").value let lastChar = UnicodeScalar("Я").value let dispatchGroup = DispatchGroup()   for charCode in firstChar...lastChar {  DispatchQueue.global().async(group: dispatchGroup) {  let docName = String(UnicodeScalar(charCode)!)  self.documentStore.createDocumenet(fromName: docName)  } }   dispatchGroup.notify(queue: DispatchQueue.main) {  print(self.documentStore) } |

Наконец, вывод в консоли будет верным.

|  |
| --- |
| 1 - В, 2 - Н, 3 - О, 4 - П, 5 - Р, 6 - С, 7 - Т, 8 - У, 9 - Ф, 10 - Х, 11 - Ц, 12 - Ч, 13 - Ш, 14 - Щ, 15 - Ъ, 16 - Ы, 17 - Ь, 18 - Э, 19 - Ю, 20 - Я, 21 - М, 22 - Г, 23 - Б, 24 - А, 25 - Д, 26 - Е, 27 - Ж, 28 - З, 29 - И, 30 - Й, 31 - К, 32 - Л, |

# Домашнее задание

1. Создать сервис для получения ленты новостей из ВК.
2. Создать класс для представления новостей.
3. Вынести работу с сетью и парсинг в глобальную очередь.

# Дополнительные материалы и используемая литература

1. [GCD на habrahabr.ru](https://habrahabr.ru/post/320152/)

# [Dispatch на developer.apple.com](https://developer.apple.com/documentation/dispatch)