МНОГОПОТОЧНОСТЬ (мультитрейдинг) –

Нужна для того чтобы все задачи выполнялись без глюков, правильно нагружая процессор и получая от него максимум пользы.

В iPhone используется 1 процессор. В современных iPhone в среднем по 6 ядер на процессоре.

Многопоточность vs Асинхронность

Многопоточность – это когда все потоки находятся в общем адресном пространстве приложения и делят одну и ту же виртуальную память. Это несколько потоков, работающих параллельно.

Асинхронность – это способ, которым будет выполняться задача

QOS (quality of service) Работает с Thread , GCD, Operation

Классы предназначены чтобы выразить намерения представленной задачи. Можно менять QOS от задачи к задаче.

* User\_interactive - используется для задач, связанных со взаимодействием с пользователем, таких как анимации, обработка событий, обновление интерфейса. По приоритету ниже чем main поток.
* User\_initiated - используется для задач, которые требуют немедленного фидбека, но не связанных с интерактивными UI событиями.
* Default - используется для задач, которые по умолчанию имеют более низкий приоритет, чем user interactive и user initiated задачи, но более высокий приоритет, чем utility и background задачи. Default является QOS по умолчанию.
* Utility - используется для задач, в которых не требуется получить немедленный результат, например запрос в сеть. Наиболее сбалансированный QOS с точки зрения потребления ресурсов.
* Background - имеет самый низкий приоритет и используется для задач, которые не видны пользователю, например синхронизация или очистка данных.
* Unspecified - используется только в том случае, если мы работаем со старым API, который не поддерживает QOS

Поток (Thread) – работа с потоком. Низкоуровневый инструмент

Потоки – последовательность инструкций которые могут быть выполнены во время выполнения приложения. Может выполнять только одну операцию в определенный момент времени. Потоков может быть только сколько ядер в процессоре.

Процесс – это наше приложение. Имеет хотя бы один поток.

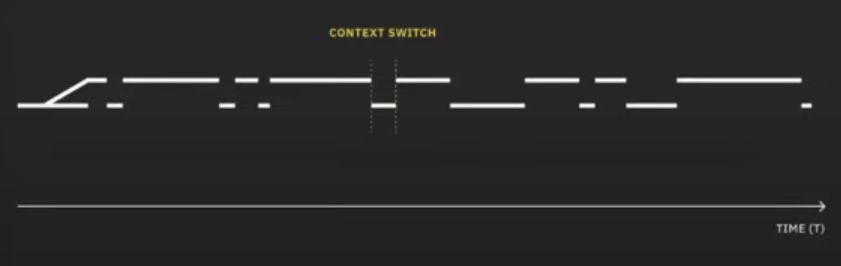
Благодаря потокам нам доступна имитация параллельной работы, т.е на 1 ядре который не может физически выполнять 2 задачи одновременно можем выполнять на нем задачи имитируя одновременную работу, так как время распределяется порционно. Представлен ниже чем GCD как сущность. Работа с потоком ведется редко, чаще с очередями GCD и задачами Operation.

Процессы и потоки нужны для того чтобы пользователю казалось что приложение может выполнять много задач параллельно.

Потоки бывают:

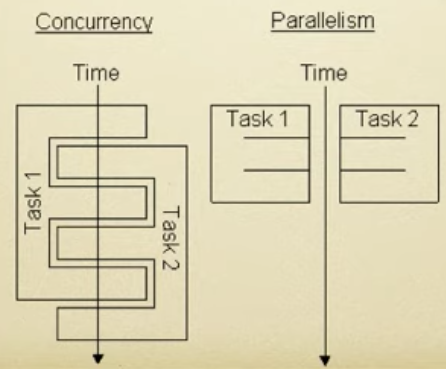
Concurrency – поток выполняется конкурентно. Если мы запускаем потоков больше чем у нас ядер на процессоре будет запускаться принцип конкуренции. Создает эффект для пользователя как будто все работает параллельно, но на самом деле он выполнит какую то часть задачи из первого потока, потом переключиться и так будет происходить пока все задачи не завершаться.

Context switch – операция переключения в принципе Concurrency (конкуренции), она является очень затратной для процессора. Если будет много потоков приложение будет зависать потому что будет много Context switch которое съест много процессорного времени вместо реальной работы.



Как решить проблемы с Context switc?

В основном это требуется на сложных, больших проектах создаются отдельно потоки, штук 5-6 для конкретных задач, например: для работы с базой данных, с сетью, для работы с Ui, рендеринг картинок и т.д. Все это создается чтобы уменьшить Context switch в конкурентной среде.

Parallelism – поток выполняется параллельно, если у устройства два ядра и мы запускаем два потока, то задачи будут выполняться параллельно.

Команды для работы с потоком:

* Start – запуск потока;
* Cancel – остановка потока;
* isMainThread (bool) – главный ли это поток?;
* isFinished (bool) – финишировал ли поток?;
* setThreatPriority (int) – установка приоритета, где 2 выше чем 1.

Высокоуровневые инструменты: GCD, Operation

API – GCD (Grand Central Dispatch) – работа с очередями

Диспетчек для работы в коде с многопоточной средой в iOS. Главный класс DispatchQueue.

Порядок помещения задачи в очередь:

1. Определяем очередь, в которую хотим отправить задание:

DispatchQueue.global / DispatchQueue.main Определяем приоритет очереди:

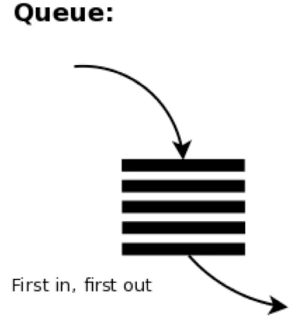
QOS

1. Определяем синхронно или асинхронно выполняем задачу:

.async / .sync

Эта технология:

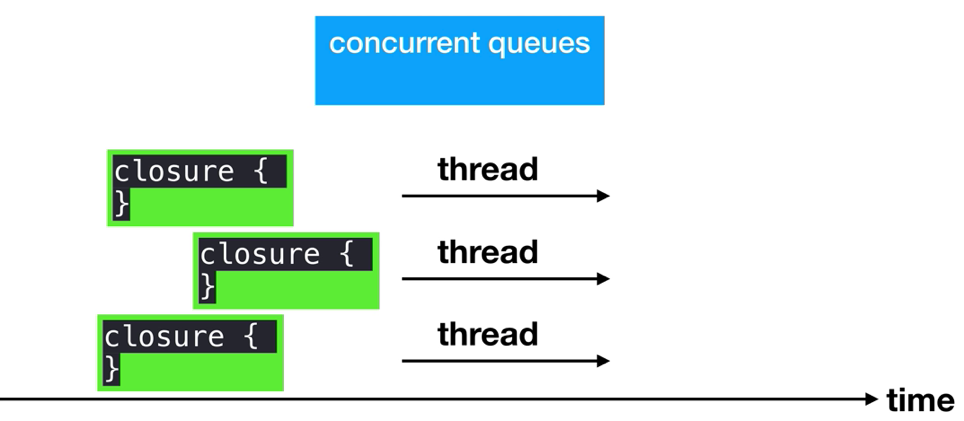
* Которая позволяет выполнять задачи асинхронно;
* Отвечает за менеджмент потоков;
* Планирует выполнение задач на потоках в зависимости от загруженности системы.
* Queue очередь – это последовательность замыканий (closures), добавленных в очередь выполнение (FIFO). К разным очередям можно привязывать разное количество кода, которое хотим выполнить = создаем кложур и отправляем его на необходимую очередь. Мы не думаем о потоке, мы сосредоточены на очередях и задачах, потоком мы больше НЕ управляем. Очередь не равно поток, исключение главный поток. Очередь использует один или несколько потоков для выполнения поступающих к ней задач.

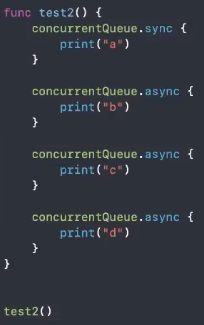


Очередь бывает 2х типов:

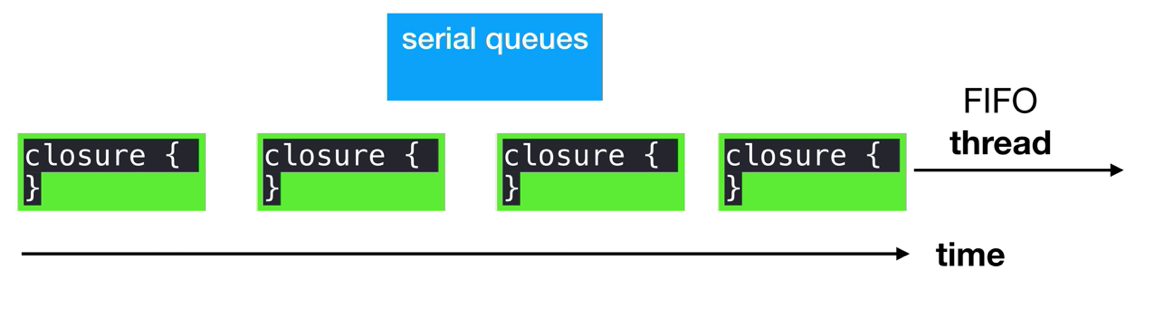
* Concurrent queue (параллельная) – чем больше добавим задач, тем медленнее будет все работать, потому что механизм будет между ними переключаться. Пример: если мы нажали одновременно на большое количество кнопок все

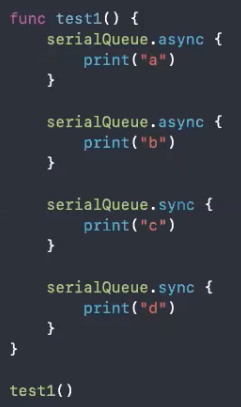
будет выполняться параллельно по чуть-чуть. Чтобы создать эту очередь нужно это явно прописать в attributes: .concurrent. С помощью Concurrent создается многопоточность в GCD.



В данной задаче первым выводится а, потому что она sync, дальше идут async но уже в какой последовательности решит xcode.

* Serial queue (последовательная) – все последовательно обрабатывается, даже если есть несколько потоков. Следующий не можем выполниться пока не выполнится предыдущий. Пример: если мы нажали одновременно на большое количество кнопок все будет выполняться последовательно. Не ровна потоку, очередь использует поток или несколько потоков для выполнения поступающих к ней задач. На Serial не бывает Context Switch, ибо нет переключений между потоками.

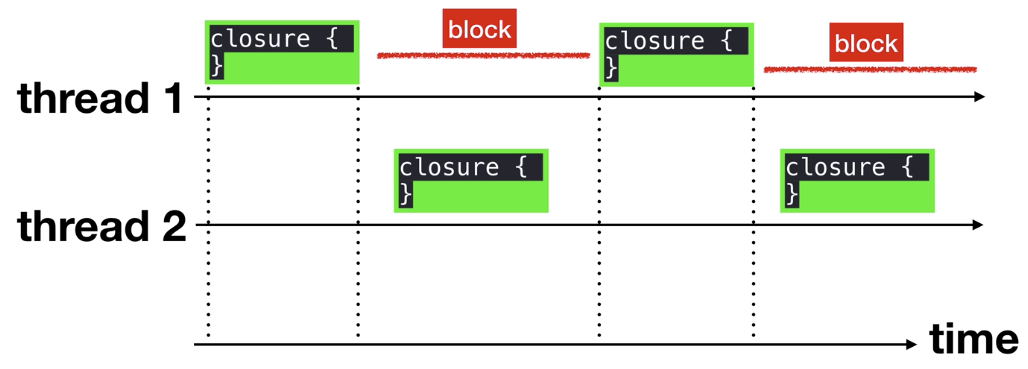


В задаче выведется: а, b, c, d потому что все выполняется в Serial и тут неважно Sync или Async.

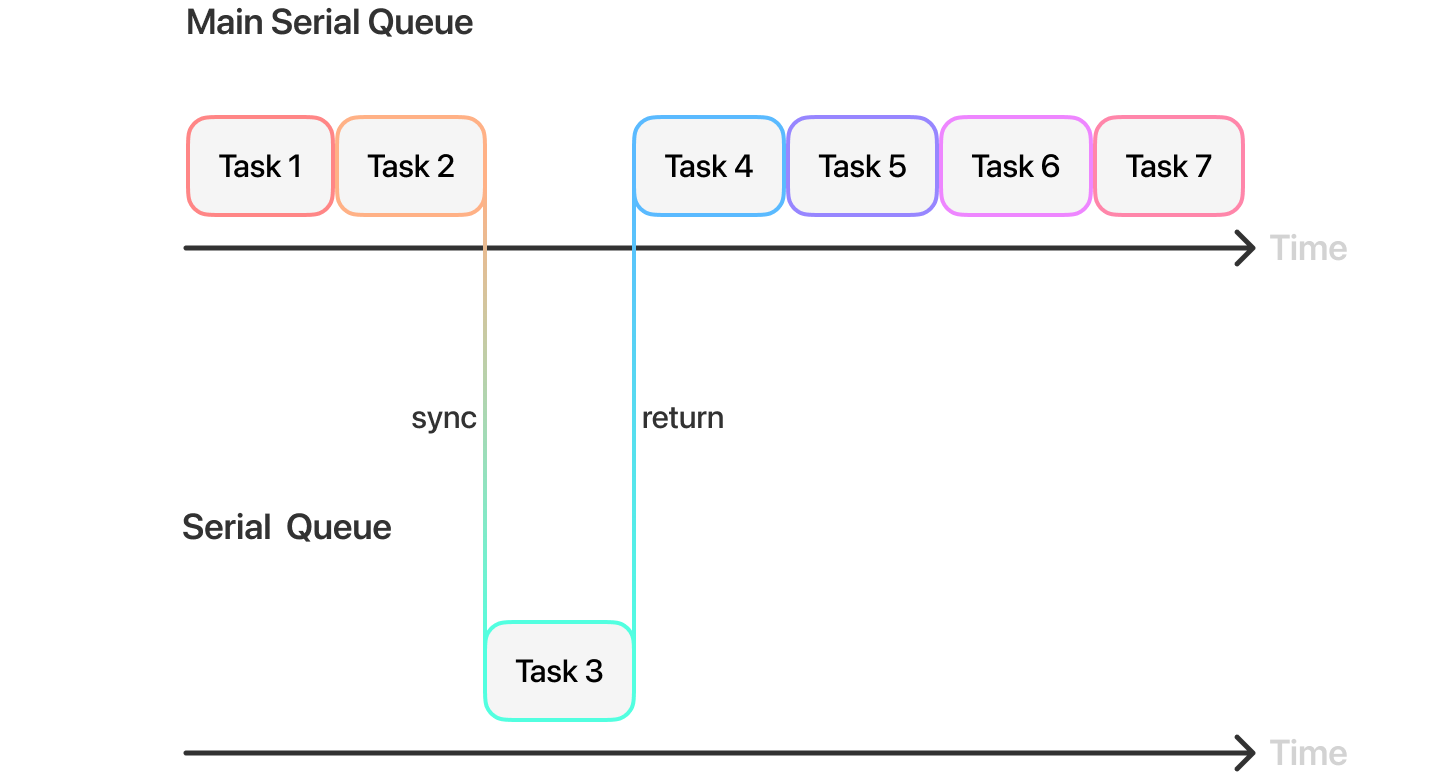
У Serial и Concurrent есть Sync или Async выполнение задач

Serial

* Sync – подразумевает что текущая очередь будет заблокирована до момента пока задача которую мы отправили на другую очередь не будет выполнена. DispatchQueue.main.sync вызывает Deadlock. В данном примере хоть и использованно два потока все равно на выполнение задач уйдет больше времени чем в Async потоке.

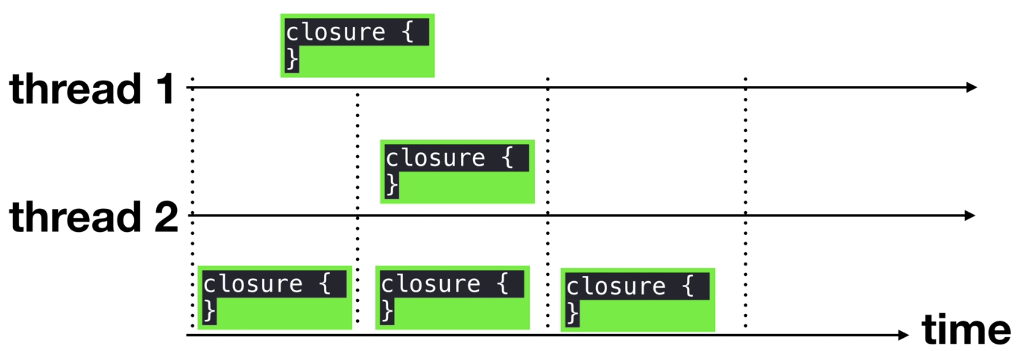


Как Serial работает с Sync:

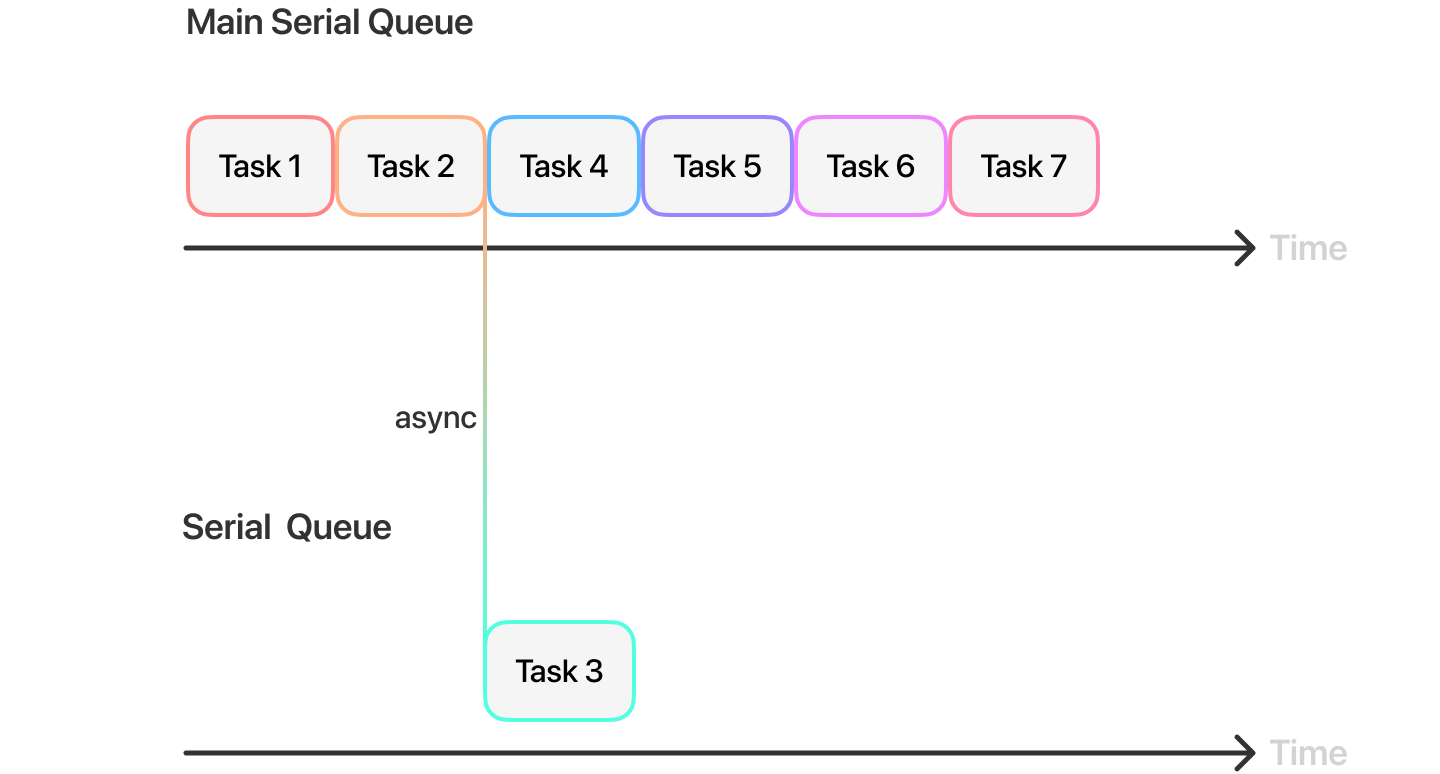


Main поток блокируется и ожидает выполнение задачи 3 на другом потоке.

* Async – при отправки задачи на другую очередь мы не блокируемся на текущей. За меньшее количество времени выполняется больше задач.



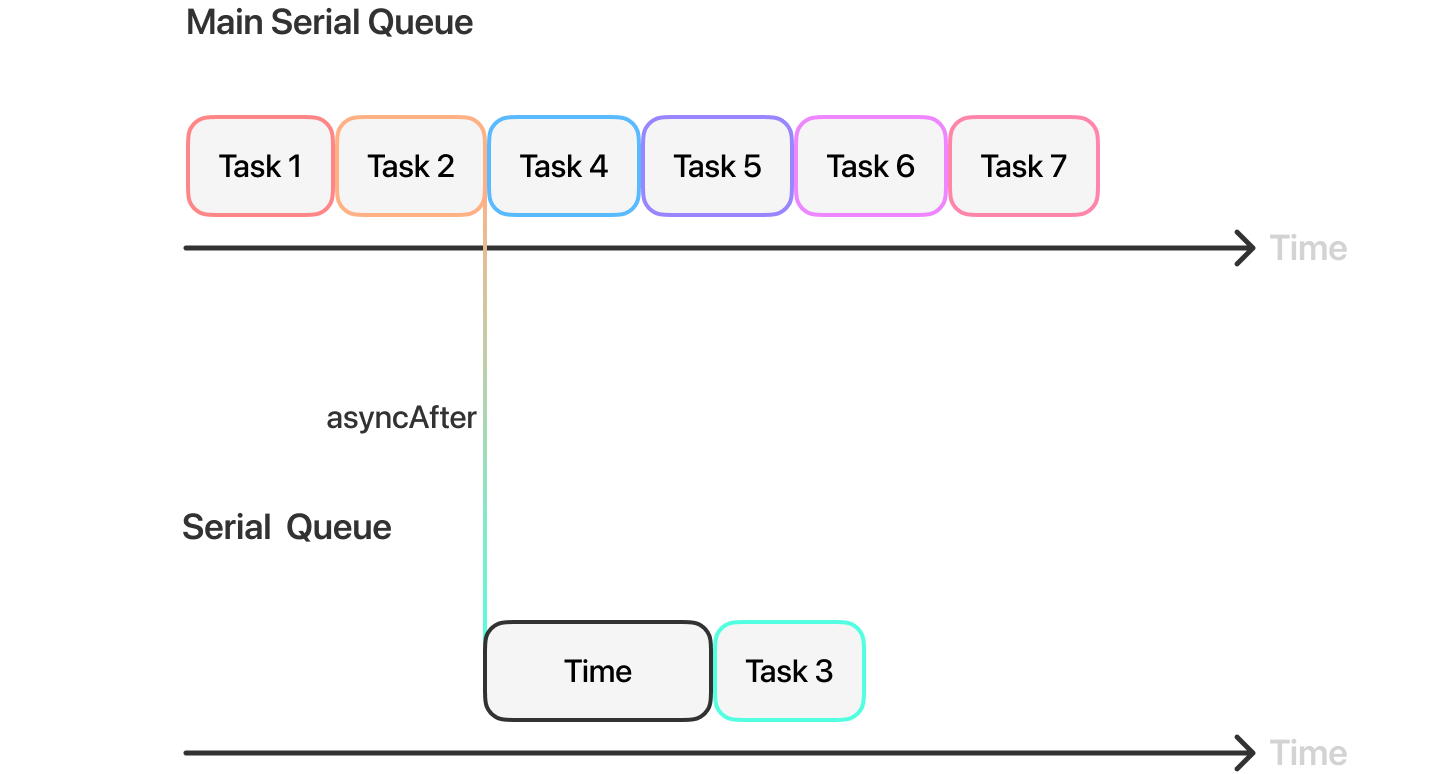
Как Serial работает с async:



Main поток НЕ блокируется и НЕ дожидается выполнение задачи 3 на другом потоке.

Async After

Метод, позволяющий отложить асинхронное выполнение задачи на определенное время. Данный метод идентичен async, за исключением аргумента deadline.



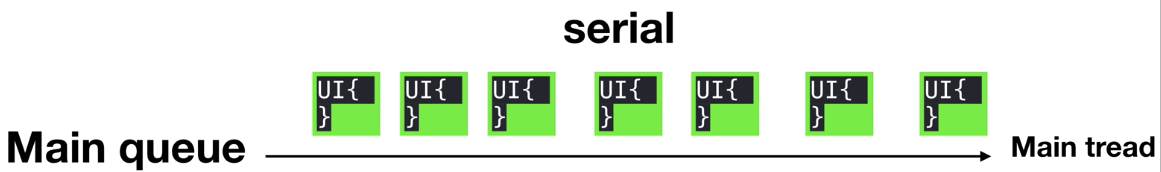
Concurrent



В целом sync и async работают также как и с serial.

Системные очереди: 5 штук

Main Queue – это главная очередь в приложении, все что связано с UI должно обязательно выполняться в этой очереди. Это serial очередь (все задачи выполняются по очереди – последовательная очередь). Работает до того, как мы начнем что либо делать в приложении. Main Queue работает в Main Thread потоке. Можно ли UI запускаем на не в Main? Нет, потому что многие элементы не потокобезопасны в UIKit, т.е можно поймать ошибку Race Condition в условной TableView. Еще нельзя потому что UI в приоритете.



* Global Queue – concurrent очереди которые разделяются всей системой. Есть 4 таких очереди с разными приоритетами: high, default, low, background, но сейчас используется QOS.
* Custom Queue – создаваемые очереди которые могут быть serial и сoncurrent.

Dispatch work item – надсройка над задачей

Инкапсулирует работу, которая должна быть выполнена в очереди отправки или группе отправки. В основном он используется в сценариях, где нам требуется возможность отсрочки или отмены выполнения блока кода. Это позволяет нам отменить задачу, поставленную в очередь, однако мы можем отменить задачу только до того, как она достигнет начала очереди и начнет выполняться. В нем также можно указывать QOS.

Dispatch Group

Объект, позволяющий объединить cхожие задачи в группу и синхронизировать их поведение. Группа позволяет присоединить к ней несколько задачь или DispatchWorkItem и запланировать их асинхронное выполнение на одной или нескольких очередях. Когда все задачи в группе будут выполнены, группа уведомит об этом какую-либо очередь и выполнит на ней completion handler. Так же группа позволяет нам дождаться выполнения задач в группе синхронно, без использования уведомления. Работает как с Main так и с Сoucurrent.

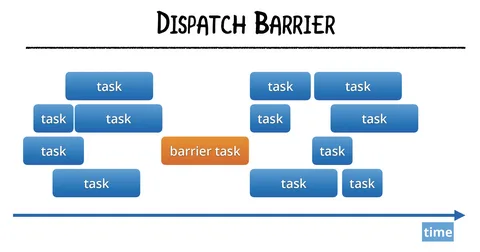
Средства синхронизации – для того чтобы избежать ситуации когда несколько потоков неожиданно изменяют какие-то данные можно использовать средства синхронизации.

Бывают:

* Semaphore; - GCD
* Mutex; - Thread
* Barrier; - GCD, раньше был аналог Read / Write lock
* Recursive mutex. - Thread

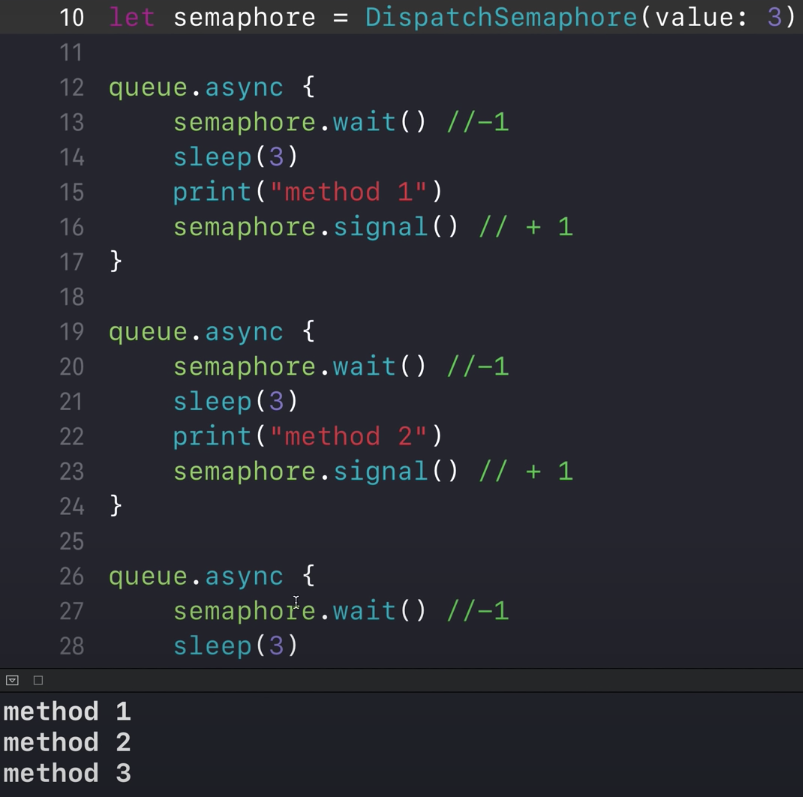
Dispatch barrier – GCD

Механизм синхронизации задач в очереди. Не дает другим потокам влиять на основной. Для того, чтобы добавить барьер, необходимо передать соответствующий флаг в метод async. Когда мы добавляем барьер в параллельную очередь, она откладывает выполнение задачи, помеченной барьером (и все остальные, которые поступят в очередь во время выполнения такой задачи), до тех пор, пока все предыдущие задачи не будут выполнены. После того, как все предудщие задачи будут выполнены, очередь выполнит задачу, помеченную барьером самостоятельно. Как только задача с барьером будет выполнена, очередь вернется к своему нормальному режиму работы. Решает проблему - Race condition (Состояние гонки)



Simaphore – GCD

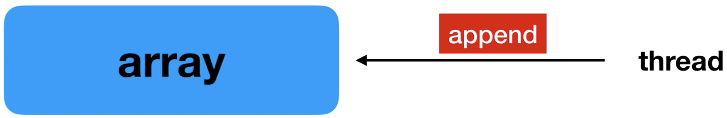
Базовый инструмент синхронизации в GCD. Semaphore позволяет нам ограничить количество потоков, которые могут единовременно обращаться к очереди. Для этого необходимо передать количество потоков в инициализатор класса DispatchSemaphore. Помимо ограничения количества потоков, семафор позволяет блокировать очередь до тех пор, пока не будет вызван метод signal. Методы signal (+1) и wait (-1) работают по принципу инкрементирования / декрементирования внутреннего каунтера семафора (аналогично рекурсивному mutex). Это означает, что поток будет разблокирован только тогда, когда каунтер равен значению value, которое мы передаем в инициализатор. Может работать с большим количеством потоков сразу. Отличается от Mutex тем что может работать с большим количеством потоков а Mutex только с одним

Если вводим в value: 1 то, выведется сначало №1 через 3 секунды №2 и еще через 3 №3. Если откроем на вход сразу value 3 то выведется все 3 задачи одновременно. Если в value передать 0 то semaphore перестанет кого-либо пропускать, чтобы разбудить очередь создаем semaphore.signal

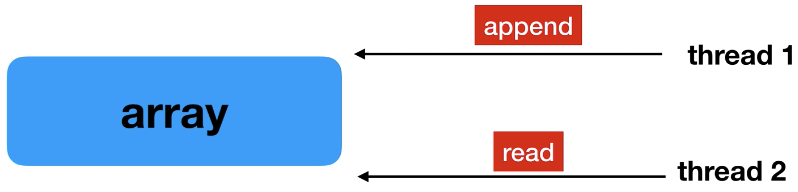
Mutex – защита объекта от других потоков – Thread

Работа с Mutex написанная на «C» работает в 15 раз быстрее чем GCD или Semaphore.

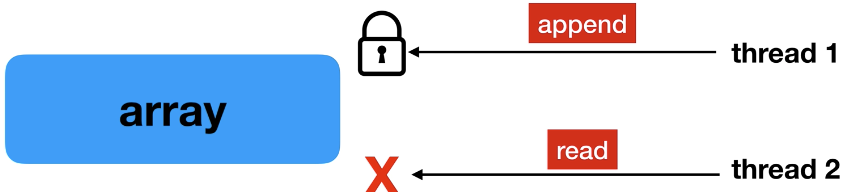
Пример: есть созданный массив в главном потоке, потом создаем второй ассинхронный поток и хотим в нем записывать или считывать данные в массив.



Создаем еще один поток. Один главны, второй нет. Один поток пытается что-то добавить а второй считать у нас получается коллизия данных (искажение данных)

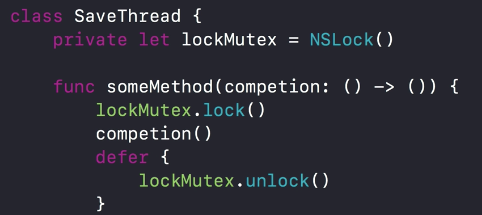


Для этого используется синхронизация, т.е мы говорит потокам становится в очередь если они хотят что-то сделать с данными



* Если потоку №1 который защищен Mutex нужен будет доступ к данным потока №2 то этот поток «засыпает» пока Mutex не будет освобожден. Mutex может блокировать и освобождать объект.

Работа c Mutex



Lock – закрыть

Defer – нужен чтобы если что-то случиться с программой произошел unlock

Unlock – открытие

* Решает проблему - Race condition (Состояние гонки)

Recursive mutex – Thread

Разновидность базового mutex, которая позволяет потоку захватывать ресурс множество раз до тех пор, пока он не освободит его. Ядро операционной системы сохраняет след потока, который захватил ресурс и позволяет ему захватывать ресурс повторно. Рекурсивный мьютекс считает количество блокировок и разблокировок, таким образом ресурс будет захвачен до тех пор, пока их количество не станет равно друг другу. Чаще всего используется в рекурсивных функциях.

Фреймворк Operation Queue – работа с задачами

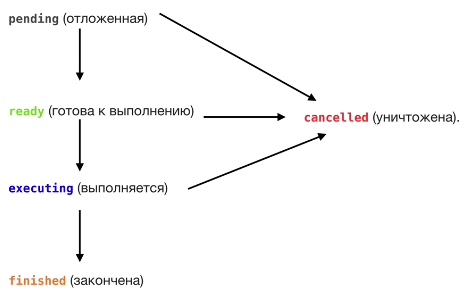
Opetation – абстрактный класс в котором объединяются наши данные и код. Своего рода оболочка над кодом, который должен быть выполнен в отдельном потоке. Объединение кода и данных называют задачей. Является потоко безопастной структурой.

Opetation Queue – класс объединяющий наши задачи в единый объект в котором происходит управление их жизненным циклом. Очередь управляет помещенными задачами на основе их приоритета и готовности. Чтобы задача была в потоке ее нужно передать в Opetation Queue. В 99% задач работает асинхронно. FIFO.

Надстройка на GCD написанная на Obg-c, работает по принципу FIFO. Обычно используется для более сложных задач. Главный класс Operation Queue. Различия между Operation и GCD:

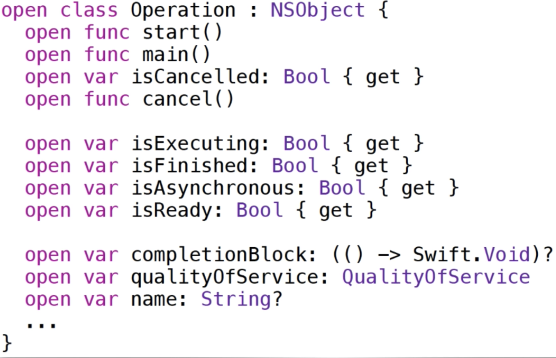
* В GCD можно отменить задачу перед тем как она начнет выполняться (до помещения в очередь), Operation позволяет это сделать из 3-х положений;
* GCD работает несколько быстрее.

Жизненный цикл Operation:



На первых трех этапах мы можем отменить задачу.

Методы Operation:



Проблемы многопоточности:

* Race condition (Состояние гонки) – ситуация, возникающая, когда время или порядок событий влияют на корректность результата т.e при каждом запуске кода разный результат. Такие проблемы, как правило, очень сложно отладить, так как проблема может возникать случайно и практически невозможно подобрать правильные шаги для воспроизведения ошибки. Эту ситуацию можно решить с помощью барьера или Mutex;
* Priority inversion (Инверсия приоритетов) – это ситуация когда низкоприоритетный поток выполняет свою работу раньше чем высокоуровневый. Т.е важные задачи не выполняются в отличии от мелких.
* Deadlock (Взаимная блокировка) - когда два потока пытаются перехватить блокировку друг у друга и получается что они ждут друг друга бесконечно и программа при этом не работает. Это можно сделать вызвав метод sync на main очереди (последовательной) все это приводит к взаимной блокировки = Deadlock. Нельзя – DispatchQueue.main.sync
* Livelock (Активная блокировка) – потоки перекидываются работой но при этом никто эту работу не делает и получается что система работает, но в холостую
* Starvation (Голодание) – проблема, когда параллельный процесс не может получить ресурсы, необходимые для выполнения его работы. Livelock считается под случаем Starvation, так как параллельные процессы одинаково «голодают», и никакая работа не выполняется до конца. Под голоданием обычно подразумевают когда не все, а один или несколько потоков мешают одному или нескольким другим потокам выполнять их работу так, как это задумывалось разработчиком (то есть наиболее эффективно).
* Data Race (Гонка за данными) – состояние, когда один поток обращается к изменяемому объекту, в то время как другой поток записывает в него.

Run Loop

Run Loop (цикл выполнения) - является механизмом, который позволяет потокам обрабатывать события (events) бесконечно в любое время.

Доп. вопросы на собеседовании

* Асинхронность возможна без многопоточности!
* Можем ли мы работать с UI не в main очереди?

Нет, потому что многие элементы не потокобезопасны в UIKit, т.е можно поймать ошибку Race Condition в условной TableView. Еще нельзя потому что UI в приоритете.

* Чем отличаются sync / async от параллелизма?

Синхронность и асинхронность это способ взаимодействия между очередями, а параллелизм это концепция на которой строятся все устройства поддерживающие параллелизм.

* Concurrency и Concurrent queue – это разные вещи. Одно это принцип, а второе это тип queue, который лишь частично реализовывает этот принцип.