



Concurrencia iOS

Miguel Mexicano Herrera



Que es concurrencia

La concurrencia es la simultaneidad de hechos.

- Un programa concurrente es aquel en el que ciertas unidades de ejecución internamente secuenciales (procesos o threads), se ejecutan paralela o simultáneamente.



Procesos Síncronos y Asíncronos

Cuando en la ejecución de un programa cada tarea se ejecuta después de que ha terminado otra tarea anterior estamos hablando de procesos síncronos.

Cuando en un proceso una tarea no tiene que esperar a que se termine de ejecutar otra tarea para continuar estamos hablando de procesos asíncronos.

Un programa en ejecución, o proceso, puede funcionar en modo unithread: los subprocessos o tareas que lo forman crean su propio hilo, destruyendo al anterior. Todo va paso a paso.

Pero un proceso también puede funcionar en modo multithread: cada subprocesso posee su propio hilo, de manera que pueden ejecutarse varias tareas al mismo tiempo, lo que se llama concurrencia.



GCD - Grand Central Dispatch

GCD es una capa abstracción que está construida por encima de la capa de los threads y nos sirve para realizar operaciones tanto asíncronas como síncronas

pertenece al lenguaje de código abierto swift

cada tarea es colocada en una cola

la tarea debe tener inicio y fin

se crea una tarea a través de un closure

colas serializadas y colas concurrentes



Main Thread

es serializado

tiene 5 colas globales que ejecutan la UI



Colas Serializadas

Ejecutan las tareas en orden, una tras otra.

se agregan mediante un FIFO(First input first output)

Usan un mismo hilo.



Colas Concurrentes

trabaja con varios hilos.

No espera a que otra tarea inicie para empezar

su ejecución depende de los recursos del sistema

se pueden agregar restricciones para hacer que la tarea sea serializada



Tipos de colas GCD y funcionamiento

Cola serializadas del hilo principal

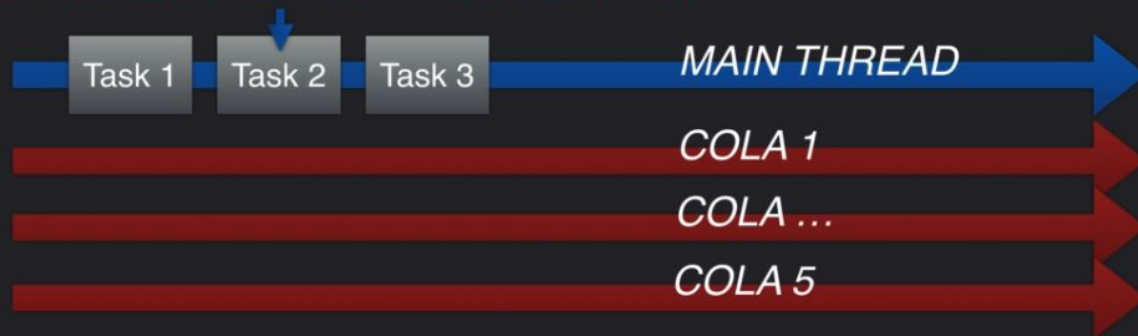
Cola serializada sincrona en hilo secundario

Cola concurrente asincrona en hilos secundarios

Nota: Evitar el trabajo en el hilo principal salvo que se trabaje con la UI

Las tareas pesadas se deben llevar a cabo en una cola concurrente asíncrona, en un hilo secundario y cuando acaben puede darse una respuesta en el hilo principal

1. COLA SERIALIZADA DEL HILO PRINCIPAL (GCD)



- ▶ El hilo principal ejecuta las tareas de forma síncrona y serializada.
- ▶ La tarea 2 espera a que la 1 finalice.
- ▶ Las tareas se ejecutan secuencialmente, una tras otra: serializadamente.
- ▶ El hilo principal tiene 5 colas, pero solo sirven para introducir tareas serializadas.

2. COLA SERIALIZADA SÍNCRONA EN HILO SECUNDARIO (GCD)



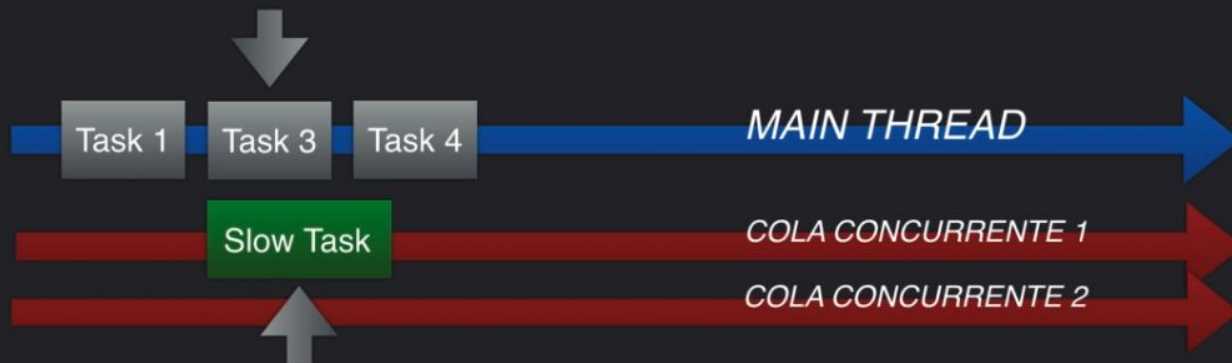
- ▶ El hilo principal funciona en modo serializado y síncrono.

3. COLA CONCURRENTE ASÍNCRONA EN HILOS SECUNDARIOS (GCD)



- ▶ Tenemos 3 tareas en una cola concorrente.
- ▶ El sistema tiene n hilos disponibles y GCD los asignará de forma automática.
- ▶ Si no hay hilos disponibles porque están ocupados con una tarea, se crea uno nuevo.

BUENAS PRÁCTICAS EN TAREAS PESADAS



- Movemos entonces la tarea lenta a una cola concurrente, para que salga del hilo principal y se ejecute a la vez, pero sin parar a la tarea 3 ni a la UI.



Ejemplo Progress Bar

`DispatchQueue.global().async`

`DispatchQueue.main.async`



Closure

Los **closures** son bloques de funcionalidad, son muy similares a los bloques en C y Objective-C y a las lamdas en otros lenguajes de programación.

Los **closures** pueden capturar y almacenar referencias de cualquier constante o variable del contexto en el cual fueron definidas.

CLOSURES

- ▶ Un *closure* para definir una tarea es algo tan simple como esto:

```
let closure = {  
  print("Hola")  
}
```

```
var x = 10
```

```
let closure = {  
  x += 1  
  print("Hola")  
}
```


- ▶ Ambos son de tipo () -> Void, pero el ejemplo de la izquierda no está capturando (encerrando) ningún valor fuera de su ámbito. Solo usa elementos creados dentro de él.
- ▶ El *closure* de la derecha usa x que está creado fuera de su ámbito. Para que este pueda ejecutarse en cualquier momento, debe encerrar a x para asegurar que este exista cuando vaya a ejecutarse. Por eso un *closure* es un cierre.



Retención de memoria

Weak: Las referencias débiles en Swift son aquellas que no afectan el recuento de retención de las instancias, por lo que **no pueden proteger al objeto de la desasignación del ARC. (self opcional)**

Unowned: son utilizadas cuando la instancia que tiene la referencia débil tiene el mismo ciclo de vida que la otra instancia. A efectos prácticos, esto significa que el objeto **B** siempre va a estar en memoria mientras el objeto **A** también lo esté (y además, no tiene sentido que no sea así).(desempaquetado implícito) self.algo!



```
1 import UIKit
2
3 let closure1 = {
4     print("Hola")
5 }
6
7 var x = 10
8
9 let closure2 = {
10     x += 1
11     print("Valor de x \ \(x)")
12 }
13
14 sleep(2)
15
16 closure2()
17
18
```

Ejemplo closures que escapan y no

```
var array:[() -> ()] = []

class Test {
    var x = 10
    private func noEscapa(completion: () -> ()) {
        completion()
    }

    private func escapa(completion: @escaping () -> ()) {
        array.append(completion)
    }

    func testSync() {
        noEscapa {
            x = 20
        }
        noEscapa(completion: { x = 21 })
        escapa {
            print("Hola")
        }
        escapa { [unowned self] in
            self.x = 15
            print(self.x )
        }
    }
}

do {
```



Problemas de concurrencia

Race Condition: acceso a recursos en

Inversión de la prioridad: cambiar la prioridad

deadlock: las tareas de bloquean mutuamente



funciona con singleton DispatchQueue

cada tarea en la cola es un dispatchworkitem y se puede agrupar en dispatch groups

enviar a la cola con async and asyncAfter, sync



QOS Quality of Service

prioriza las tareas

userInteractive : interacción de usuario refresh UI o animaciones

userInitiated: acción inicializada por el usuario que requiere una respuesta rápida pero no en la interfaz grabar un documento o tocar la interfaz

utility: tardará un tiempo indeterminado y no requiere resultado inmediato (subida de datos al server)

background: Tarea en 2 plano copia de seguridad

default: entre user initiated y utilities

unspecified: no se especifica la prioridad



```
DispatchQueue.main.async {
    print("Hola main")
}

DispatchQueue.global().async {
    print("Hola global")
}

DispatchQueue.global(qos: .userInteractive).asyncAfter(deadline: .now() + 5, execute: {
    print("Global concurrente pasados 5 segundos")
})

let dispatch = DispatchQueue(label: "com.banregio.serial")
let dispatchC = DispatchQueue(label: "com.banregio.concurrent", qos: .userInteractive, attributes:
    .concurrent)

dispatchC.schedule(after: .init(.now() + 86400), tolerance: .seconds(1)) {
    print("Acción programada para mañana")
}

dispatch.sync {
    print("Hola")
}

DispatchQueue.concurrentPerform(iterations: 10) { index in
    print("Iteración \(index)")
}
```



Dispatch Group

```
import Foundation

DispatchQueue.global(qos: .userInteractive).async {
    let group = DispatchGroup()

    DispatchQueue.global().async(group: group) {
        print("Tarea de ejecución")
    }

    group.enter()
    print("Hola Grupo 1")
    group.leave()

    func load(delay: UInt32, completion: () -> ()) {
        sleep(delay)
        completion()
    }

    group.enter()
    load(delay: 2) {
        print("Hola Grupo 2")
        group.leave()
    }

    group.notify(queue: .main) {
        print("Tareas terminadas")
    }
}
```