Concurrency, Operation Queue and Grand Central Dispatch

ddas.tech/concurrency-operation-queue-and-grand-central-dispatch/

September 19, 2022

Created By: Debasis Das (Sep 2022)

In this article we will cover the following topics in Cocoa with examples and sample code in Swift depicting the usage of each.

What is Concurrency?

- Doing multiple things at the same time.
- Taking advantage of number of cores available in multicore CPUs.
- Running multiple programs in parallel.

Objectives of Concurrency

- Running program in background without hogging CPU.
- Define Tasks, Define Rules and let the system take the responsibility of performing them.
- Improve responsiveness by ensuring that the main thread is free to respond to user events.
- Leverage more cores to do more work in the same amount of time.

Problems with the Threaded Approach

- Threads are low level tool that needs to be managed manually.
- Creating a correct threading solution is difficult.
- Thread synchronization adds complexity to the project

- Incorrectly implemented threading solution might make the system even worse
- Scalability is an issue when it comes to utilizing multiple available cores.

When to Use Threads?

- Threads are still a good way to implement code that must run in real time
- Dispatch Queues make every attempt to run their tasks as fast as possible but they do not address the real time constraints

Operations and Operation Queue

- Object oriented way to encapsulate work that needs to be performed asynchronously
- An Operation object is an instance of NSOperation(abstract) class.
- NSOperation class has two concrete subclasses that can be used as is
    - NSInvocationOperation (used to execute a method)
    - NSBlockOperation (used for executing one or more blocks concurrently)
- An operation can be executed individually/manually by calling its start method or it can be added to an OperationQueue.

NSInvocationOperation

- A class we can use as-is to create an operation object based on an object and selector from your application.
- We can use this class in cases where we have an existing method that already performs the needed task. Because it does not require subclassing, we can also use this class to create operation objects in a more dynamic fashion.

NSBlockOperation

- A class we use as-is to execute one or more block objects concurrently.
- Because it can execute more than one block, a block operation object operates using a group semantic; only when all of the associated blocks have finished executing is the operation itself considered finished.

Grand Central Dispatch

- Grand Central Dispatch provides queues to which the application can submit tasks in the form of block objects.
- The block of code submitted to dispatch queues are executed on a pool of threads managed by the system. Each task can either be executed synchronously or asynchronously.
- In case of synchronous execution the program waits for the execution to be finished before the call returns. In case of asynchronous call the method call returns immediately.

- **Dispatch Queues** can be serial or concurrent. In case of serial dispatch queues the work items are executed one at a time and in case of concurrent although the tasks are dequeued in order but they run in parallel and can finish in any order.
- **Main Queue** – When the app is launched the system automatically creates a special queue called as main queue. Work items on the main queue is executed serially on the applications main thread

Time to test the concepts

Lets starts by defining 3 functions that we will run in different settings in GCD.

```
// The below methods prints the thread on which they are being executed
// Main Thread or Background?
        func printApples(){
            print("printApples is running on = \(Thread.isMainThread ? "Main
Thread":"Background Thread")")
            for i in 0..<3{
                print("🍏\(i)")
            }
        }

        func printStrawberries(){
            print("printStrawberries is running on = \(Thread.isMainThread ? "Main
Thread":"Background Thread")")
            for i in 0..<3{
                print("🍓\(i)")
            }
        }

        func printBalls(){
            print("printBalls is running on = \(Thread.isMainThread ? "Main
Thread":"Background Thread")")
            for i in 0..<3{
                print("🎱\(i)")
            }
        }

//Normal Function calls would lead to the below output, All running in the main
thread
func testPrintMethods(){
            printApples()
            printStrawberries()
            printBalls()
        }
```

```
printApples is running on = Main Thread
🍏0
🍏1
🍏2
printStrawberries is running on = Main Thread
🍓0
🍓1
🍓2
printBalls is running on = Main Thread
🎱0
🎱1
🎱2
```

Using One Dispatch Queue to run 3 tasks in a background thread

Note: All the methods are running in the background thread, but as the same queue is being used they are running in serial.

```
func queueTest1(){
            let queue = DispatchQueue(label: "com.bigdecimals.queue1")
            queue.async {
                self.printApples()
            }
            queue.async {
                self.printStrawberries()
            }
            queue.async {
                self.printBalls()
            }
        }
```

```
printApples is running on = Background Thread
🍏0
🍏1
🍏2
printStrawberries is running on = Background Thread
🍓0
🍓1
🍓2
printBalls is running on = Background Thread
🎱0
🎱1
🎱2
```

Using 3 different dispatch queues to run the three functions in the background threads

```
func queueTest2(){
            let queue1 = DispatchQueue(label: "com.bigdecimals.queue1")
            let queue2 = DispatchQueue(label: "com.bigdecimals.queue2")
            let queue3 = DispatchQueue(label: "com.bigdecimals.queue3")
            queue1.async {
                self.printApples()
            }
            queue2.async {
                self.printStrawberries()
            }
            queue3.async {
                self.printBalls()
            }
        }
```

```
printApples is running on = Background Thread
printBalls is running on = Background Thread
printStrawberries is running on = Background Thread
🍏0
🍏1
🍏2
🍓0
🎱0
🎱1
🎱2
🍓1
🍓2
```

Using Dispatch Queue to run tasks on the main thread itself.

In the below code we have created dispatch queues but are specifically instructing it to run on the main thread using **sync** calls. You can see that printApple runs on the main thread and the other 2 function runs on background threads.

```
func queueTest3(){
            let queue1 = DispatchQueue(label: "com.bigdecimals.queue1")
            let queue2 = DispatchQueue(label: "com.bigdecimals.queue2")
            let queue3 = DispatchQueue(label: "com.bigdecimals.queue3")
            queue1.sync {
                self.printApples()
            }
            queue2.async {
                self.printStrawberries()
            }
            queue3.async {
                self.printBalls()
            }
        }
```

```
printApples is running on = Main Thread
🍏0
🍏1
🍏2
printStrawberries is running on = Background Thread
printBalls is running on = Background Thread
🍓0
🍓1
🍓2
🎱0
🎱1
🎱2
```

Using Global Queue

```
func queueTest4(){
            let globalQueue = DispatchQueue.global()
            globalQueue.async {
                self.printApples()
            }
            globalQueue.async {
                self.printStrawberries()
            }
            globalQueue.async {
                self.printBalls()
            }

        }
```

```
printApples is running on = Background Thread
printStrawberries is running on = Background Thread
printBalls is running on = Background Thread
🍓0
🍓1
🍓2
🍏0
🍏1
🍏2
🎱0
🎱1
🎱2
```

Below code forces to run a task on the main thread on the global queue by using globalQueue.sync

```
func queueTest5(){
            let globalQueue = DispatchQueue.global()
            globalQueue.sync {
                self.printApples()
            }
            globalQueue.async {
                self.printStrawberries()
            }
            globalQueue.async {
                self.printBalls()
            }

        }
```

```
printApples is running on = Main Thread
🍏0
🍏1
🍏2
printStrawberries is running on = Background Thread
printBalls is running on = Background Thread
🍓0
🍓1
🍓2
🎱0
🎱1
🎱2
```

Running the functions on the main queue will run all the functions on the main thread itself

```
func queueTest6(){
            let mainQueue = DispatchQueue.main
            mainQueue.async {
                self.printApples()
            }

            mainQueue.async {
                self.printStrawberries()
            }
            mainQueue.async {
                self.printBalls()
            }
        }
```

```
printApples is running on = Main Thread
🍏0
🍏1
🍏2
printStrawberries is running on = Main Thread
🍓0
🍓1
🍓2
printBalls is running on = Main Thread
🎱0
🎱1
🎱2
```

Quality of Service

In the next set of examples we will create Dispatch Queues using a thread priority using QOS.
Before we begin please read through the summary offering of the QoS parameters

- **User-interactive** used for user interaction, refreshing user interface, performing animations. The focus is on responsiveness and performance
- **User-initiated** – Work is nearly instantaneous, Opening or saving document
- **Utility** – Work that may take some time to complete and doesn't require an immediate result, such as downloading or importing data.
- **Background**– Work that operates in the background and isn't visible to the user, such as indexing, synchronizing, and backups.

In Addition to the above there are 2 more QoS

- **Default**– The priority level of this QoS falls between user-initiated and utility. Work that has no QoS information assigned is treated as default, and the GCD global queue runs at this level.
- **Unspecified** – This represents the absence of QoS information and cues the system that an environmental QoS should be inferred. Threads can have an unspecified QoS if they use legacy APIs that may opt the thread out of QoS.

For more details on Quality of Service refer to
https://developer.apple.com/library/content/documentation/Performance/Conceptual/EnergyGuide-iOS/PrioritizeWorkWithQoS.html

```
func queueTest7(){
    let queue1 = DispatchQueue(label: "com.bigdecimals.queue1", qos:
.userInteractive, attributes: .concurrent, autoreleaseFrequency: .inherit, target:
DispatchQueue.global())
    let queue2 = DispatchQueue(label: "com.bigdecimals.queue2", qos: .utility,
attributes: .concurrent, autoreleaseFrequency: .inherit, target:
DispatchQueue.global())

        queue1.async {
            self.printStrawberries()
        }
        queue2.async {
            self.printBalls()
        }

    }

/*
     Output - In the above sample. The thread priority or quality of service plays a
key role as User Interactive is of highest priority and although both the queues are
configured to run on the background thread, the print strawberries completes faster
     printStrawberries is running on = Background Thread
     printBalls is running on = Background Thread
     🍓0
     🍓1
     🍓2
     🎱0
     🎱1
     🎱2

     */

func queueTest8(){
       let queue1 = DispatchQueue(label: "com.bigdecimals.queue1", qos:.utility,
attributes: .concurrent, autoreleaseFrequency: .inherit, target:
DispatchQueue.global())
       let queue2 = DispatchQueue(label: "com.bigdecimals.queue1", qos: .background,
attributes: .concurrent, autoreleaseFrequency: .inherit, target:
DispatchQueue.global())

       queue1.async {
           self.printStrawberries()
       }
       queue2.async {
           self.printBalls()
       }

   }
```

```
/*
     Output - Utility is of higher priority than the background QoS

     printStrawberries is running on = Background Thread
     printBalls is running on = Background Thread
     🍓0
     🎱0
     🍓1
     🍓2
     🎱1
     🎱2

     */
```

Delaying the Execution of a Queue

```
func queueTest9(){
            let queue1 = DispatchQueue(label: "com.knowstack.queue1", qos:.utility,
attributes: .concurrent, autoreleaseFrequency: .inherit, target:
DispatchQueue.global())
            let queue2 = DispatchQueue(label: "com.knowstack.queue1", qos:
.background, attributes: .concurrent, autoreleaseFrequency: .inherit, target:
DispatchQueue.global())

            queue1.asyncAfter(deadline: .now()+5.0) {
                print("In Print Strawberries = \(Date())")
                self.printStrawberries()

            }

            queue2.async {
                print("In Print Balls = \(Date())")
                self.printBalls()

            }
        }

You can see that the printStrawberries was called after a delay of 5 seconds

In Print Balls = 2022-06-12 02:46:41 +0000
printBalls is running on = Background Thread
🎱0
🎱1
🎱2
In Print Strawberries = 2022-06-12 02:46:46 +0000
printStrawberries is running on = Background Thread
🍓0
🍓1
🍓2
```