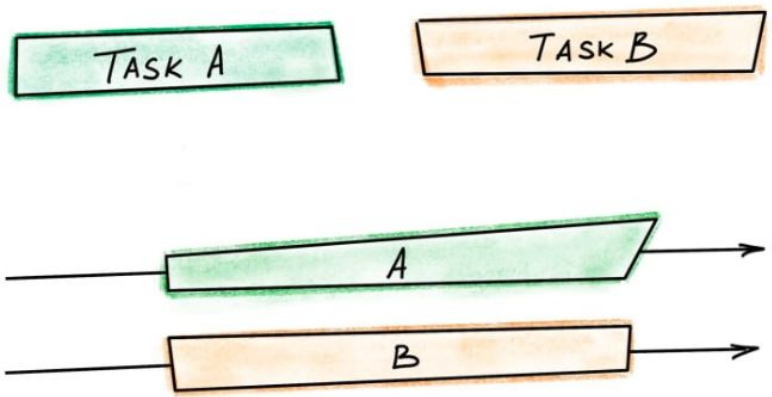
iOS Concurrency

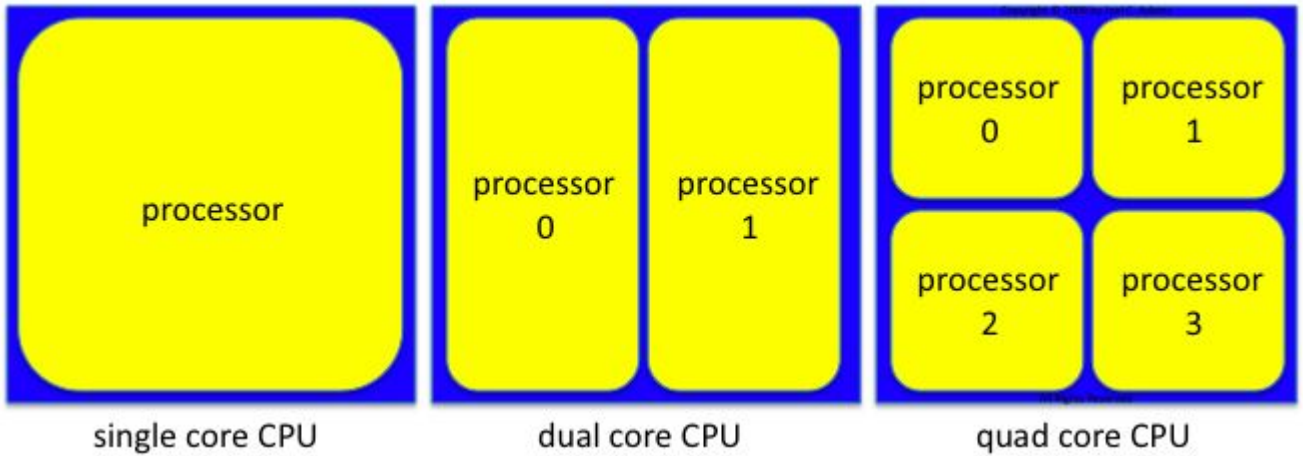
Concurrency là gì?

- Concurrency mô tả khái niệm chạy một số nhiệm vụ cùng một lúc. Điều này có thể xảy ra theo cách chia sẻ thời gian trên một lõi CPU hoặc thực sự song song nếu có nhiều lõi CPU.



Tại sao phải sử dụng Concurrency?

- Chúng ta sử dụng concurrency để làm cho mượt giao diện, tối ưu phần cứng, tốc độ tải nhanh làm cho trải nghiệm người dùng tốt hơn, tận dụng thế mạnh đa lõi của thiết bị

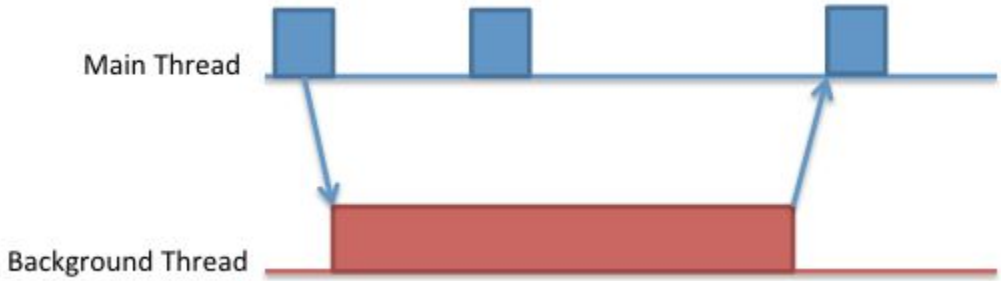


Sử dụng Concurrency như thế nào?

- Khi cấu trúc app, chúng ta chỉ định những task nào cần chạy đồng thời.
- Những task sử dụng tài nguyên dùng chung (**shared resource**) sẽ không được chạy đồng thời vì sẽ gây ra sai lệch kết quả.
- Những task truy xuất những nguồn tài nguyên khác nhau có thể chạy đồng thời.
- Trong iOS, Apple cung cấp 2 cách để hiện thực concurrency: **The Grand Central Dispatch (GCD)** và **NSOperationQueue**

Nơi nào task chạy?

- **Task** được chạy trên **thread**.
- **UI** được chạy trên **main thread** và hệ thống tạo ra những thread khác cho tất cả các task.
- Ứng dụng có thể sử dụng những thread của hệ thống hoặc tự tạo thread mới.



Thread

- Thread là một tài nguyên có giới hạn trong iOS, chúng ta chỉ được phép có 64 thread cùng một lúc. Cách để khởi tạo và thực thi một thread như sau:

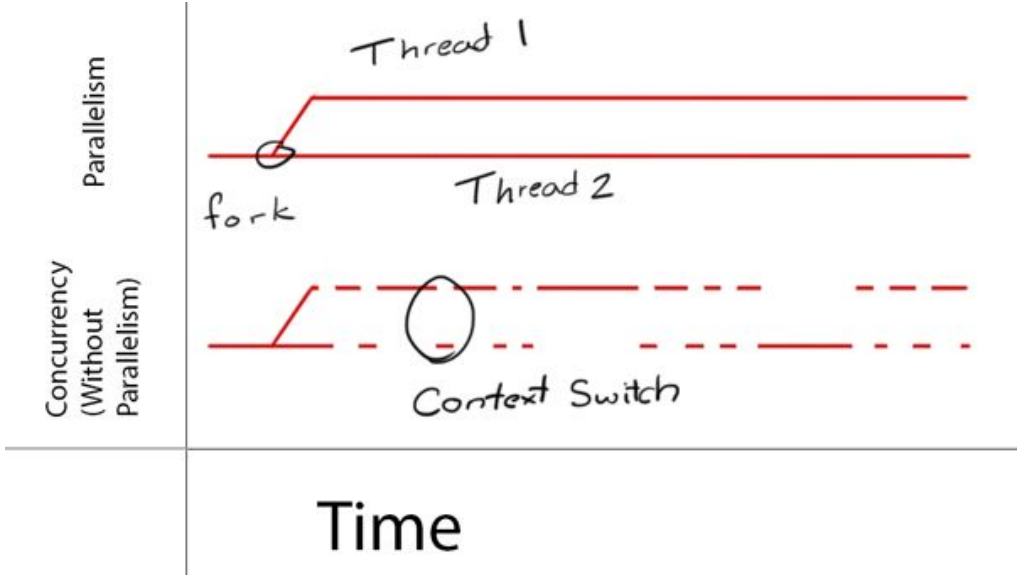
```
class CustomThread: Thread {
  override func main() {
    do_something
  }
}

let customThread = CustomThread()
     customThread.start()
```

- Khi chưa nắm rõ về hoạt động của Thread, chỉ nên sử dụng **main thread** của hệ thống

Grand Center Dispatch

Grand Center Dispatch là gì?

GCD là API thường được sử dụng để quản lý việc xử lý đồng thời và xử lý không đồng bộ (asynchronously) ở Unix level của hệ thống bằng cách cung cấp và quản lý các queue (hàng đợi) cho các task.



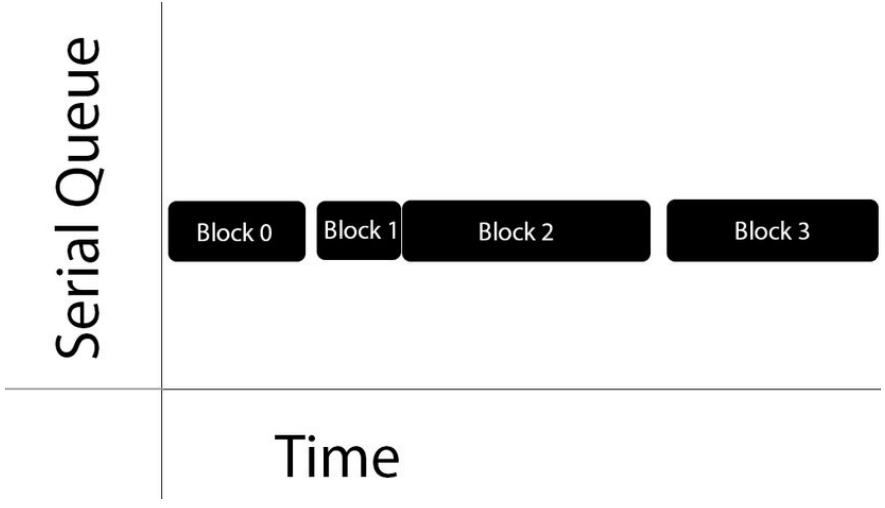
Dispatch Queue

- **Dispatch Queue** là queue được tạo ra để quản lý việc xử lý các task đồng thời hay tuần tự.
- Để sử dụng dispatch queue, chúng ta viết code dưới dạng các block, gán các block vào các dispatch queue để yêu cầu GCD xử lý.
- Có 2 loại dispatch queue: **serial queue** và **concurrent queue**

```
let queue = DispatchQueue(label: "com.test.api", qos: .background)
```

Serial Queue

- **Serial queue** là loại queue mà tại 1 thời điểm chỉ có thể chạy được 1 task.
- Khi gán các task của chúng ta vào queue này, các task sẽ được thực hiện lần lượt từng task theo thứ tự **first in first out (FIFO)**.

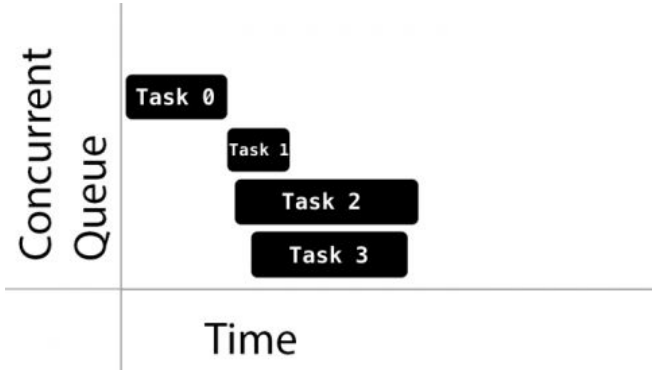


Ví dụ serial queue

```
func simpleQueue() {

    // Khai báo serial queue
    let serialQueue = DispatchQueue(label: "com.bigZero.serialQueue")

    // Vòng lặp 5 lần tượng trưng đưa 5 task vào trong serial queue
    for i in 1...5 {

        // Các task đưa vào sẽ được đánh dấu là Asynchronous
        serialQueue.async {
            print("Task \(i): - \(Thread.current))")
            // Vì hệ thống sẽ xử ý rất nhanh
            //nên mỗi lần chạy xong nghỉ 1s để quan sát rõ hơn.
            sleep(1)
        }
    }
    print("Complete! - \(Thread.current)")
}
```

Concurrent queue

- **Concurrent queue** là loại queue mà các task được xử lý song song.
- Khi các task được gán vào queue, **task nào vào trước sẽ được xử lý trước**.
- Mỗi task trong queue không phải chờ đợi task trước đó xử lý xong mới được xử lý, nghĩa là chúng ta không biết được task nào sẽ hoàn thành trước



Ví dụ concurrency queue

```
func simpleConcurrentQueue() {
    // Khai báo serial queue
    let concurrentQueue = DispatchQueue(label: "com.bigZero.concurrentQueue",
                                        attributes: .concurrent)

    // Vòng lặp 10 lần tượng trưng đưa 5 task vào trong concurrent queue
    for i in 1...10 {
        // Các task đưa vào sẽ được đánh dấu là Aynchronous
        concurrentQueue.async {
            print("Task \(i) - \(Thread.current)")
            sleep(1)
        }
    }
    print("Complete! - \(Thread.current)")
}
```

Độ ưu tiên **concurrent queue**

QoS Class	Type of work and focus of QoS	Duration of work to be performed
User-interactive	Work that is interacting with the user, such as operating on the main thread, refreshing the user interface, or performing animations. If the work doesn't happen quickly, the user interface may appear frozen. Focuses on responsiveness and performance.	Work is virtually instantaneous.
User-initiated	Work that the user has initiated and requires immediate results, such as opening a saved document or performing an action when the user clicks something in the user interface. The work is required in order to continue user interaction. Focuses on responsiveness and performance.	Work is nearly instantaneous, such as a few seconds or less.
Utility	Work that may take some time to complete and doesn't require an immediate result, such as downloading or importing data. Utility tasks typically have a progress bar that is visible to the user. Focuses on providing a balance between responsiveness, performance, and energy efficiency.	Work takes a few seconds to a few minutes.
Background	Work that operates in the background and isn't visible to the user, such as indexing, synchronizing, and backups. Focuses on energy efficiency.	Work takes significant time, such as minutes or hours.

NSOperation

Operation Queue

- **Operation queues** được xây dựng tầng bên trên của GCD. Điều này có nghĩa là chúng ta có thể sử dụng NSOperation để thực thi các task đồng thời giống như GCD
- Operation queue **không thực hiện** các task theo quy tắc **FIFO**
- Operation queue chỉ xử lý theo kiểu concurrent queue mà **không xử lý** theo kiểu **serial queue** của GCD
- Operation queue là kiểu hướng đối tượng, mỗi operation queue là một instance của lớp NSOperationQueue, và mỗi task là một instance của lớp NSOperation.

Khởi tạo operation queue

```
let operationQueue: OperationQueue = OperationQueue()
    operationQueue.addOperations([operation1], waitUntilFinished: false)
```

- Một operation có thể khởi tạo bằng nhiều cách như sử dụng block, closure, subclass. Và nếu sử dụng subclass thì đừng quên gọi đến finish, nếu không operation sẽ không bao giờ dừng lại.

```
class CustomOperation: Operation {
    override func main() {
        guard isCancelled == false else {
            finish(true)
            return
        }

        // Do something

        finish(true)
    }
}
```

- Operation là bạn có thể sử dụng dependency, như ở ví dụ dưới đây, operation1 sẽ thực thi trước operation2.

```
operation2.addDependency(operation1)

//execute operation1 before operation2
```

Độ ưu tiên của **OperationQueue**

```
public enum NSOperationQueuePriority : Int {
    case VeryLow
    case Low
    case Normal
    case High
    case VeryHigh
}
```