

# GCD

## (Grand Central Dispatch)

**Giftbot**

# History

---

- 초기에는 마이크로 프로세서의 **clock 속도를 높이는 방식으로 연산 속도를 높임**
- 점점 증가하는 전력 소비와 발생하는 열로 인해 단일 프로세서의 **clock 속도 증가에 한계를 맞게 되었으며, 이는 특히 모바일에서 영향을 크게 받음**
- 따라서 CPU 벤더들은 단일 CPU 의 클럭 속도를 개선하는 대신 여러 개의 CPU 를 탑재하는 형태로 효율을 높이는 것에 초점을 맞추게 됨
- 프로세서의 클럭 속도가 빨라지면서 자연스럽게 소프트웨어도 빨라지던 예전과 달리 멀티 코어 프로세서에서는 멀티 프로세서에게 어떻게 잘 프로그램의 동작을 배분하는 지가 중요해짐
- GCD 이전에는 멀티 스레딩을 위해 **Thread 와 OperationQueue 등의 클래스를 사용했는데, Thread 는 복잡하고 Critical Section 등을 이용한 Lock 을 관리하기 까다로웠고, OperationQueue 는 GCD 에 비해 무겁고 Boilerplate 코드들이 많이 필요한 문제가 있음**

# History

---

- GCD (Grand Central Dispatch)는 멀티 코어 프로세서 시스템에 대한 응용 프로그램 지원을 최적화하기 위해 Apple에서 개발한 기술로 스레드 관리와 실행에 대한 책임을 어플리케이션 레벨에서 운영체제 레벨로 넘김
- 작업의 단위는 Block 이라 불리며, DispatchQueue 가 이 Block 들을 관리
- GCD 는 각 어플리케이션에서 생성된 DispatchQueue 를 읽는 멀티코어 실행 엔진을 가지고 있으며, 이것이 Queue에 등록된 각 작업을 꺼내 스레드에 할당  
개발자는 내부 동작을 자세히 알 필요 없이 Queue 에 작업을 넘기기만 하면 됨
- Thread 를 직접 생성하고 관리하는 것에 비해 관리 용이성과 이식성, 성능 증가

# Grand Central Terminal



# DispatchQueue



# Migrating Away from Threads

애플 공식 문서 - Thread 클래스 대신 GCD 사용 권장

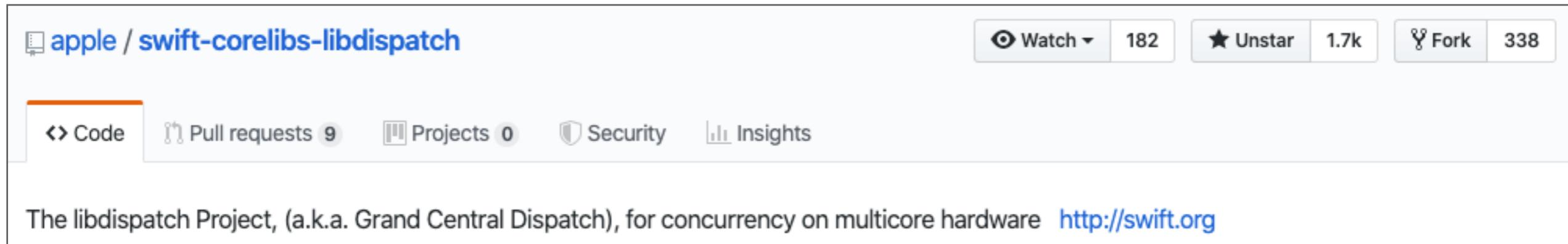
There are many ways to adapt existing threaded code to take advantage of Grand Central Dispatch and operation objects. Although moving away from threads may not be possible in all cases, performance (and the simplicity of your code) can improve dramatically in places where you do make the switch. Specifically, using dispatch queues and operation queues instead of threads has several advantages:

## GCD 의 장점

- reduces the memory penalty for storing thread stacks in the app's memory space.
- eliminates the code needed to create and configure your threads.
- eliminates the code needed to manage and schedule work on threads.
- simplifies the code

# Dispatch Framework

멀티코어 하드웨어에서 다중 작업을 동시에 수행할 수 있도록 관리하기 위한 프레임워크  
시스템에서 관리하는 Dispatch Queues에 작업을 전달하여 수행



The screenshot shows the GitHub repository page for [apple / swift-corelibs-libdispatch](#). The page includes standard GitHub metrics: Watch (182), Unstar, 1.7k forks, and 338 stars. Below the header, there are navigation links for Code (highlighted in orange), Pull requests (9), Projects (0), Security, and Insights. A descriptive text at the bottom states: "The libdispatch Project, (a.k.a. Grand Central Dispatch), for concurrency on multicore hardware <http://swift.org>".

# DispatchQueue

---

- GCD 는 앱이 블록 객체 형태로 작업을 전송할 수 있는 FIFO 대기열(Queue)을 제공하고 관리
- Queue에 전달된 작업은 시스템이 전적으로 관리하는 스레드 풀(a pool of threads)에서 실행됨
- 작업이 실행될 스레드는 보장되지 않음
- DispatchQueue 는 2개의 타입( Serial / Concurrent )으로 구분되며 둘 모두 FIFO 순서로 처리
- 앱을 실행하면 시스템이 자동으로 메인스레드 위에서 동작하는 Main 큐(Serial Queue)를 만들어 작업을 수행하고, 그 외에 추가적으로 여러 개의 Global 큐(Concurrent Queue)를 만들어서 큐를 관리
- 각 작업은 동기(sync) 방식과 비동기(async) 방식으로 실행 가능하지만 Main 큐에서는 async 만 사용 가능

# Serial Queue

Serial Queue

Time

Block 0

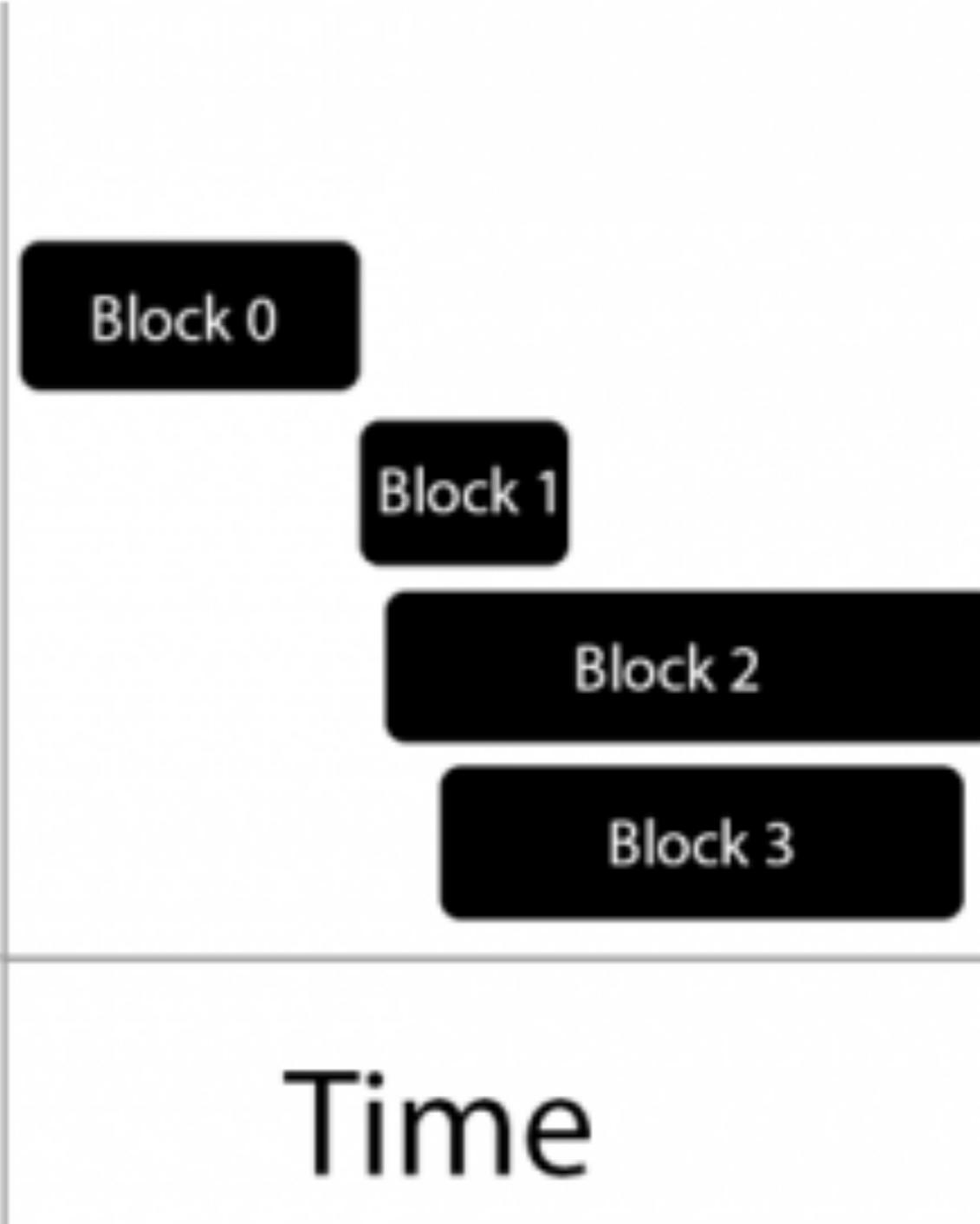
Block 1

Block 2

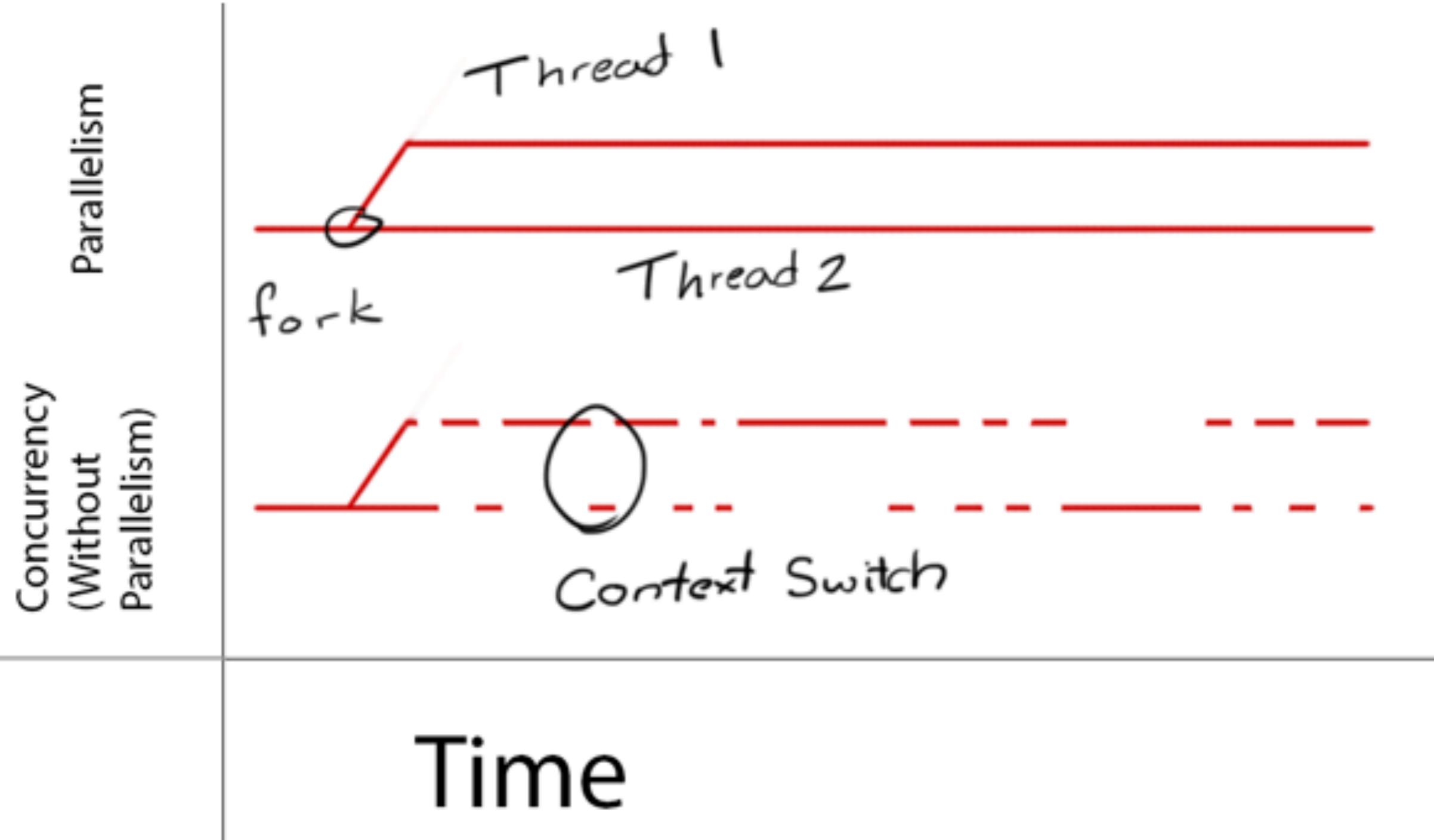
Block 3

# Concurrent Queue

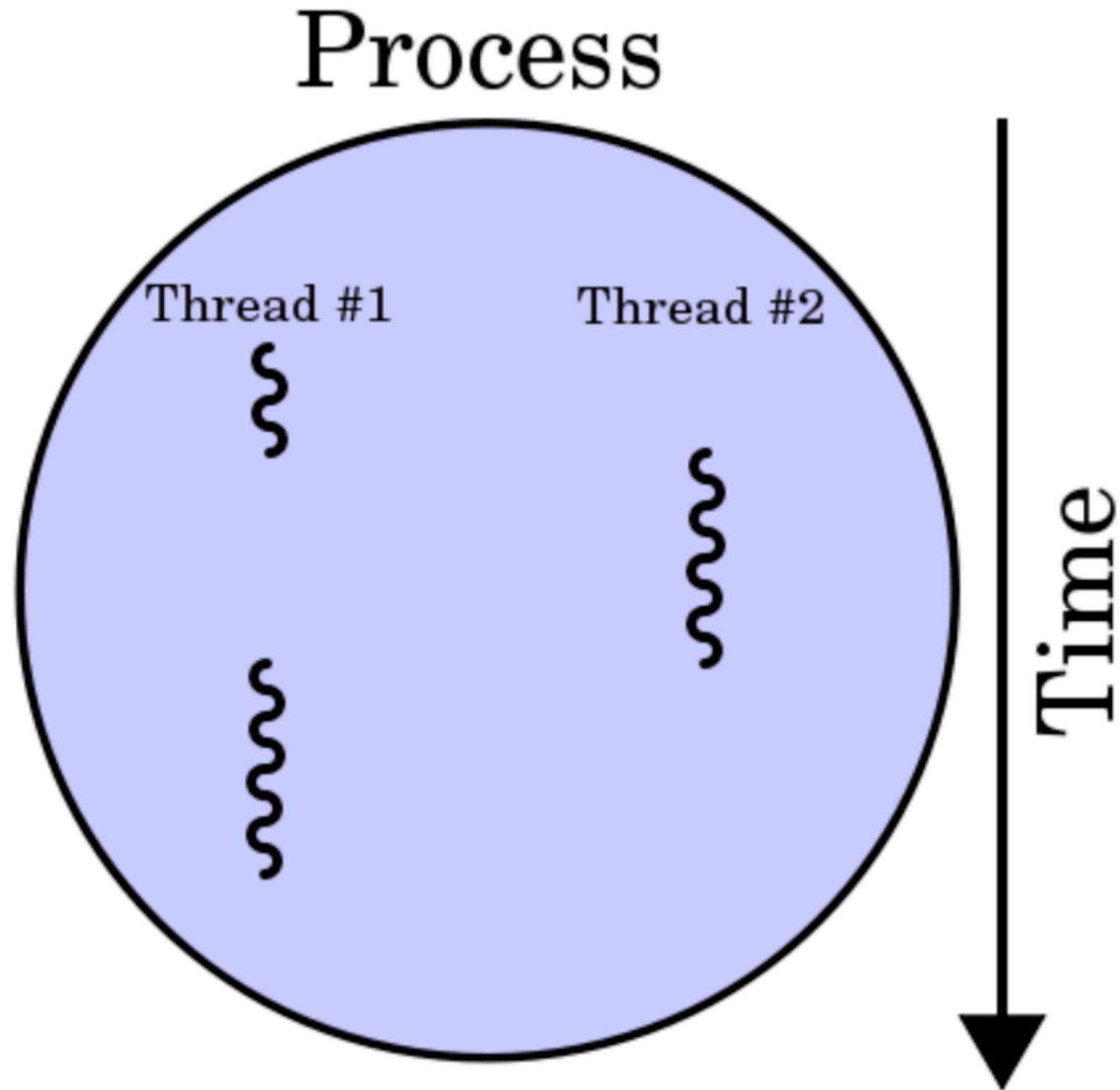
Concurrent  
Queue



# Concurrency vs Parallelism



# Concurrency (SingleCore)



# WorkItem Execution

---

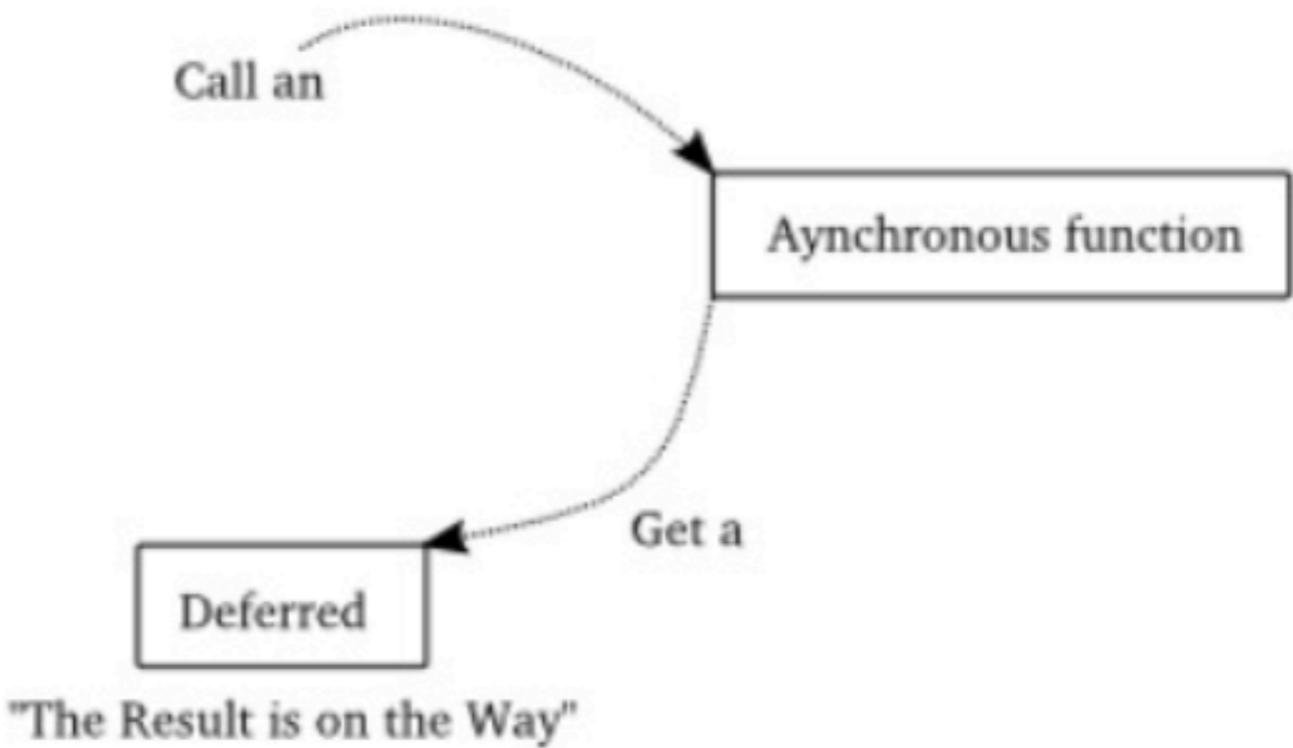
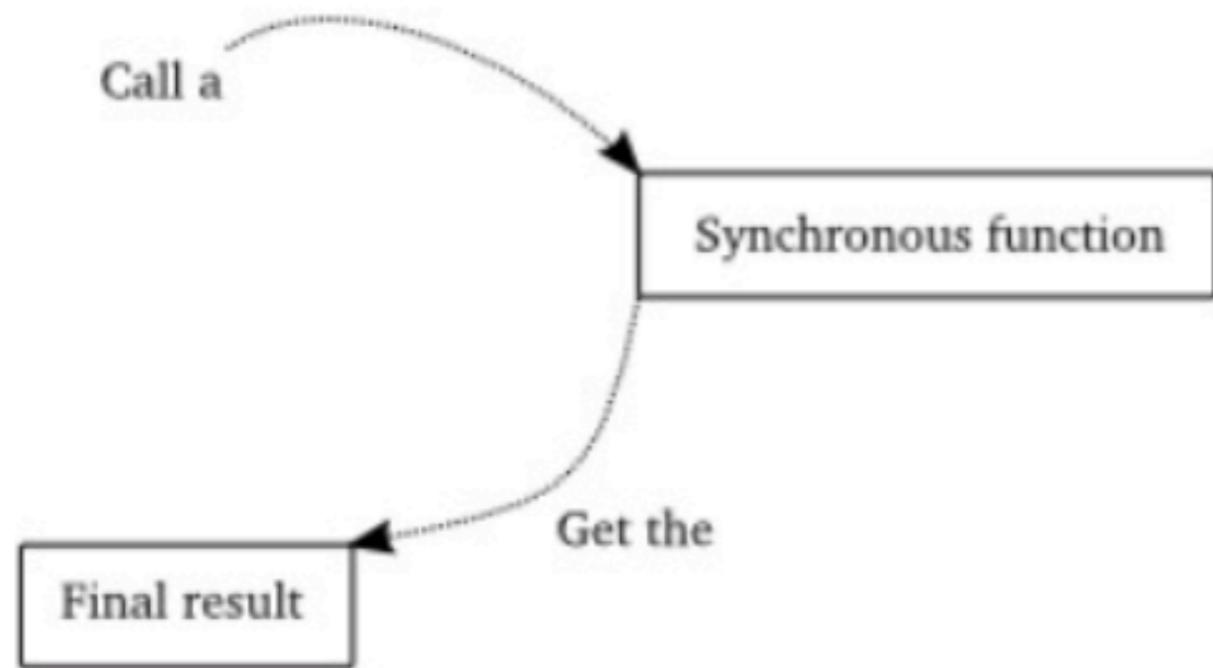
- **Synchronous**

sync 메서드를 이용해 동기적으로 실행되면, 작업이 완료될 때까지 대기한 뒤 메서드에서 return 호출  
해당 스레드의 다른 작업들은 모두 일시 정지

- **Asynchronous**

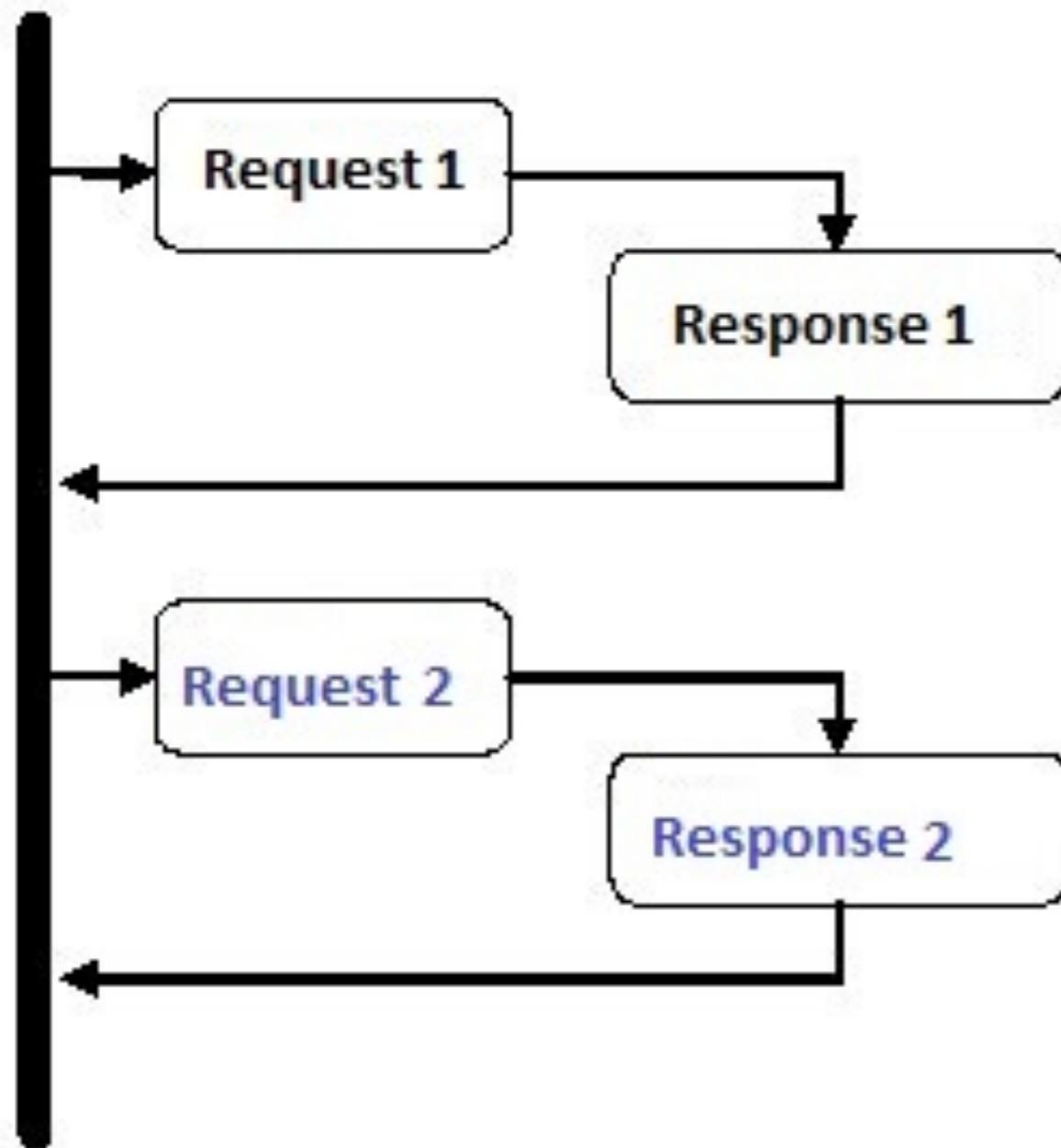
async 메서드를 이용해 비동기적으로 실행되면, 즉시 메서드에서 return 을 호출하고 작업을 수행

# Sync vs Async

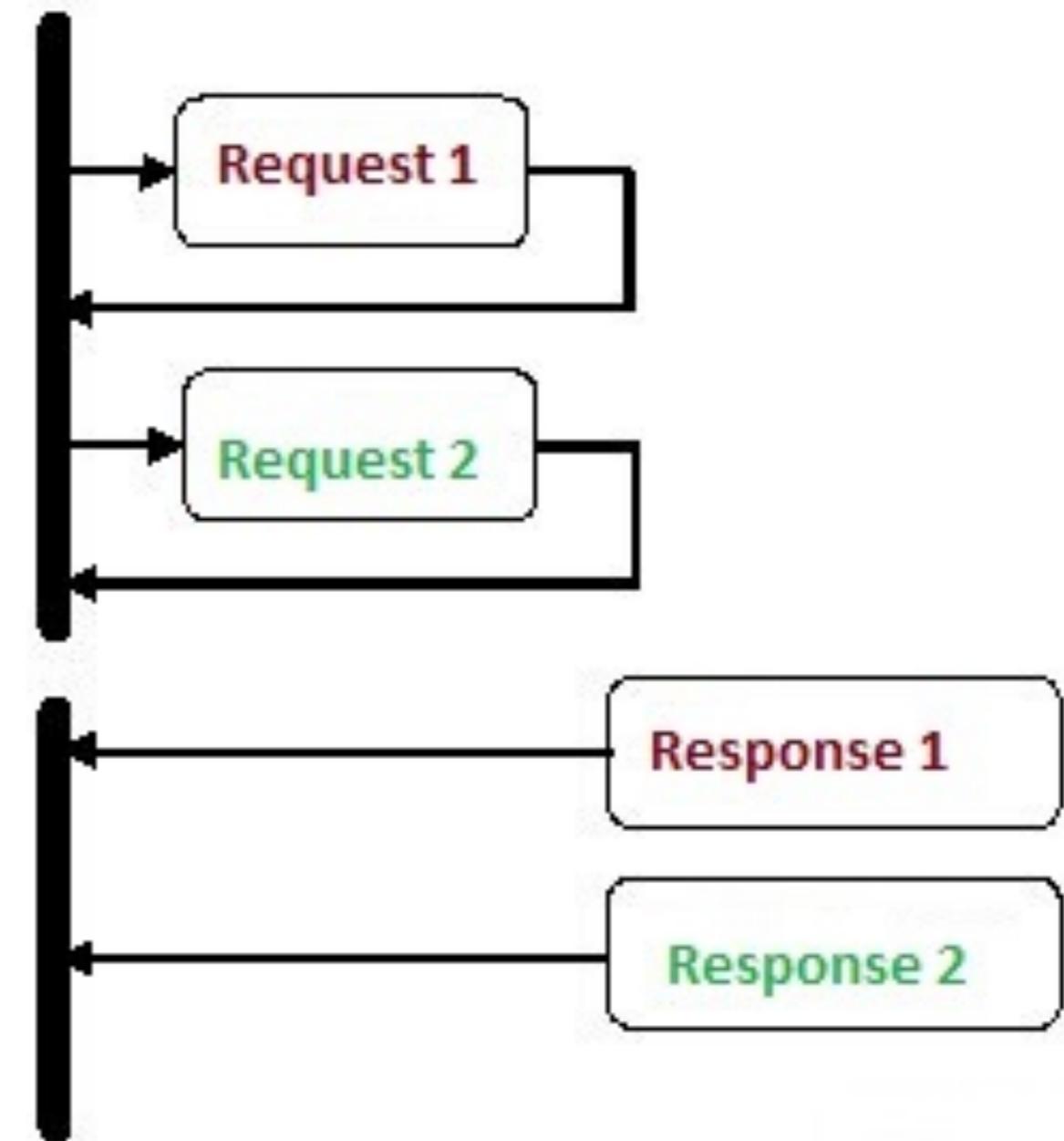


# Sync vs Async

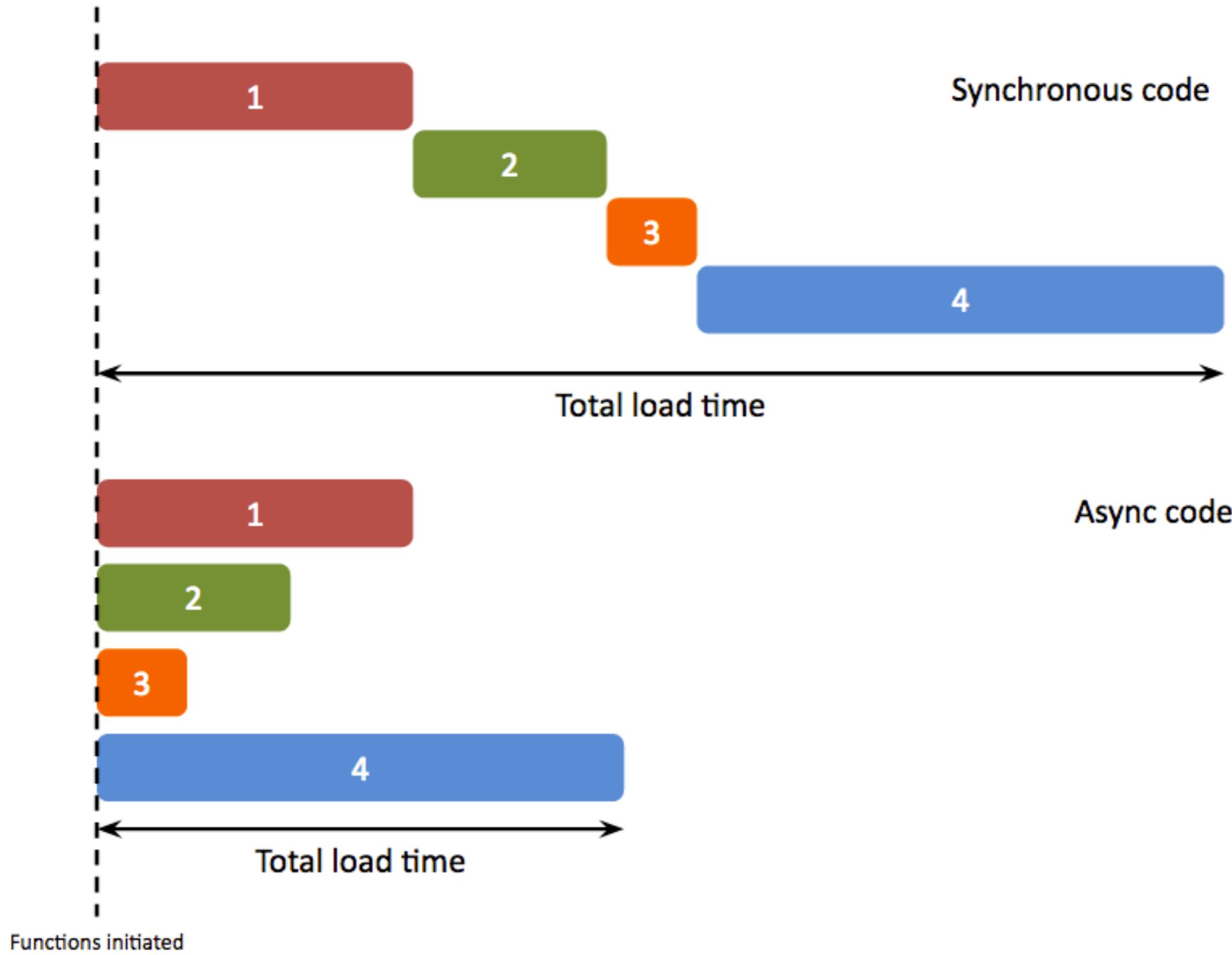
Synchronous



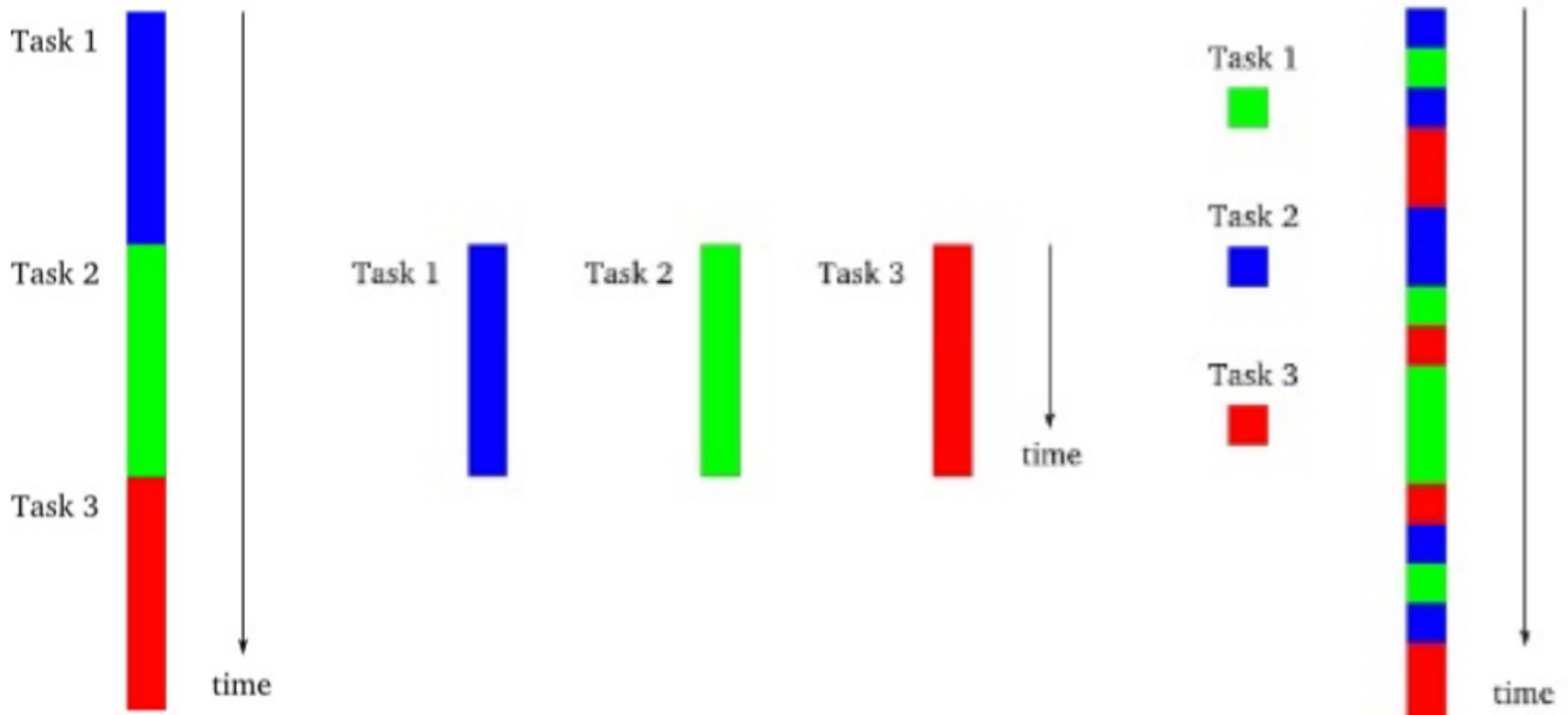
Asynchronous



# Sync vs Async



# Q. Execution Types



# System DispatchQueue

System이 제공하는 Queue는 Main 과 Global 이 있으며 앱 실행 시 생성

## [ Main ]

- UI 와 관련된 작업은 모두 main 큐를 통해서 수행하며 Serial Queue 에 해당
- MainQueue 를 sync 메서드로 동작시키면 Dead Lock 상태에 빠짐

DispatchQueue.main.async { }

## [ Global ]

- UI 를 제외한 작업에서 사용하며 Concurrent Queue 에 해당
- sync 와 async 메서드 모두 사용 가능
- QoS 클래스를 지정하여 우선 순위 설정 가능

DispatchQueue.global().async { }

DispatchQueue.global(qos: .utility).sync { }

# Custom DispatchQueue

## Non Main Thread에서 작업 수행

**Serial / Concurrent Queue 및 QoS 등의 여러 옵션을 지정하여 생성 가능**

```
public convenience init(  
    label: String,  
    qos: DispatchQoS = default,  
    attributes: DispatchQueue.Attributes = default,  
    autoreleaseFrequency: DispatchQueue.AutoreleaseFrequency = default,  
    target: DispatchQueue? = default  
)
```

# QoS (Quality of Service)

시스템은 QoS 정보를 통해 스케줄링, CPU 및 I/O 처리량, 타이머 대기 시간 등의 우선 순위를 조정

총 6개의 QoS 클래스가 있으며 4개의 주요 유형과 다른 2 개의 특수 유형으로 구분 가능

- userInteractive
- userInitiated
- default
- utility
- background
- unspecified

↑ 높음

```
public struct DispatchQoS : Equatable {  
    public let relativePriority: Int  
  
    public static let background: DispatchQoS  
    public static let utility: DispatchQoS  
    public static let `default`: DispatchQoS  
    public static let userInitiated: DispatchQoS  
    public static let userInteractive: DispatchQoS  
    public static let unspecified: DispatchQoS  
}
```

↓ 낮음

# Primary QoS classes

우선 순위가 높을 수록 더 빨리 수행되고 더 많은 전력을 소모

수행 작업에 적절한 QoS 클래스를 지정해주어야 더 반응성이 좋아지며, 효율적인 에너지 사용이 가능

## User Interactive

- 즉각 반응해야 하는 작업으로 반응성 및 성능에 중점
- main thread에서 동작하는 인터페이스 새로 고침, 애니메이션 작업 등 즉각 수행되는 유저와의 상호작용 작업에 할당

## User Initiated

- 몇 초 이내의 짧은 시간 내 수행해야 하는 작업으로 반응성 및 성능에 중점
- 문서를 열거나, 버튼을 클릭해 액션을 수행하는 것처럼 빠른 결과를 요구하는 유저와의 상호작용 작업에 할당

## Utility

- 수초에서 수분에 걸쳐 수행되는 작업으로 반응성, 성능, 그리고 에너지 효율성 간에 균형을 유지하는데 중점
- 데이터를 읽어들이거나 다운로드 하는 등 작업을 완료하는데 어느 정도 시간이 걸릴 수 있으며 보통 진행 표시줄로 표현

## Background

- 수분에서 수시간에 걸쳐 수행되는 작업으로 에너지 효율성에 중점. NSOperation 클래스 사용 시 기본 값
- background에서 동작하며 색인 생성, 동기화, 백업 같이 사용자가 볼 수 없는 작업에 할당
- 저전력 모드에서는 네트워킹을 포함하여 백그라운드 작업은 일시 중지

# Special QoS Classes

일반적으로, 별도로 사용할 일이 없는 특수 유형의 QoS

## Default

- QoS 를 별도로 지정하지 않으면 기본값으로 사용되는 형태이며 User Initiated 와 Utility 의 중간 레벨
- GCD global queue 의 기본 동작 형태

## Unspecified

- QoS 정보가 없으므로 시스템이 QoS 를 추론해야 한다는 것을 의미

QoS Class	Description
Default	<p>The priority level of this QoS falls between user-initiated and utility. This QoS is not intended to be used by developers to classify work. Work that has no QoS information assigned is treated as default, and the GCD global queue runs at this level.</p>
Unspecified	<p>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.</p>

# DispatchQueue.Attributes

**.concurrent** - Concurrent Queue 로 생성. 이 옵션 미 지정시 Serial Queue 가 기본값

**.initiallyInactive** - Inactive 상태로 생성. 작업 수행 시점에 activate() 메서드를 호출해야 동작

```
extension DispatchQueue {  
    public struct Attributes : OptionSet {  
        public static let concurrent: DispatchQueue.Attributes  
        public static let initiallyInactive: DispatchQueue.Attributes  
    }  
}
```