

---

# Thread

---

강사 주영민

# Thread

---

- 스레드(thread)는 어떠한 프로그램 내에서, 특히 **프로세스 내에서 실행되는 흐름의 단위를 말한다**. 일반적으로 한 프로그램은 하나의 스레드를 가지고 있지만, 프로그램 환경에 따라 둘 이상의 스레드를 동시에 실행할 수 있다. 이러한 실행 방식을 멀티스레드(multithread)라고 한다.

# iOS Thread

---

- 모든 app은 기본적으로 **main thread**를 가지고 있다.
- use UIKit classes only from your app's main thread.
- 기본적인 UI 및 모든 행동은 main thread에서 실행된다.

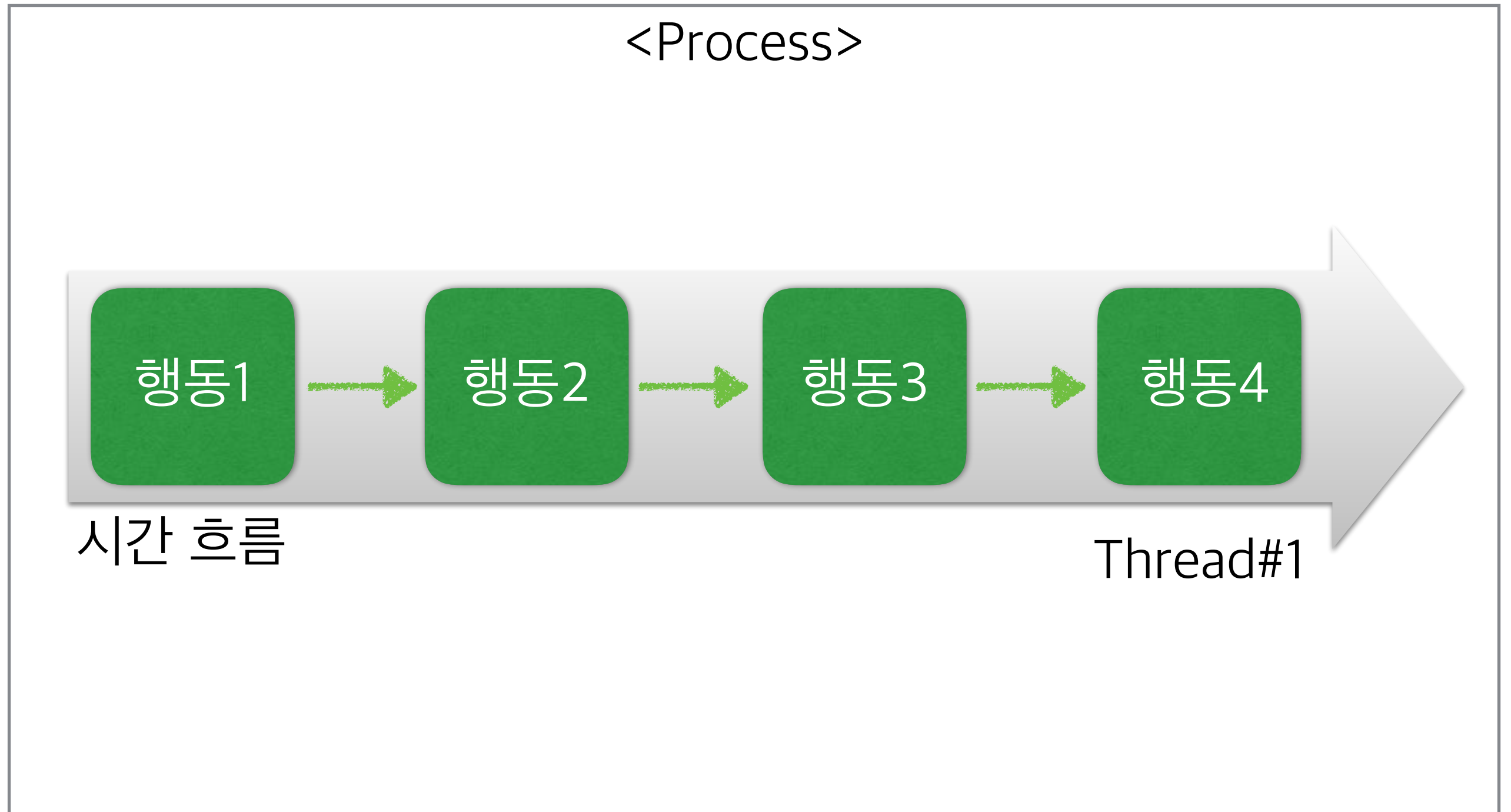
# iOS Thread

---

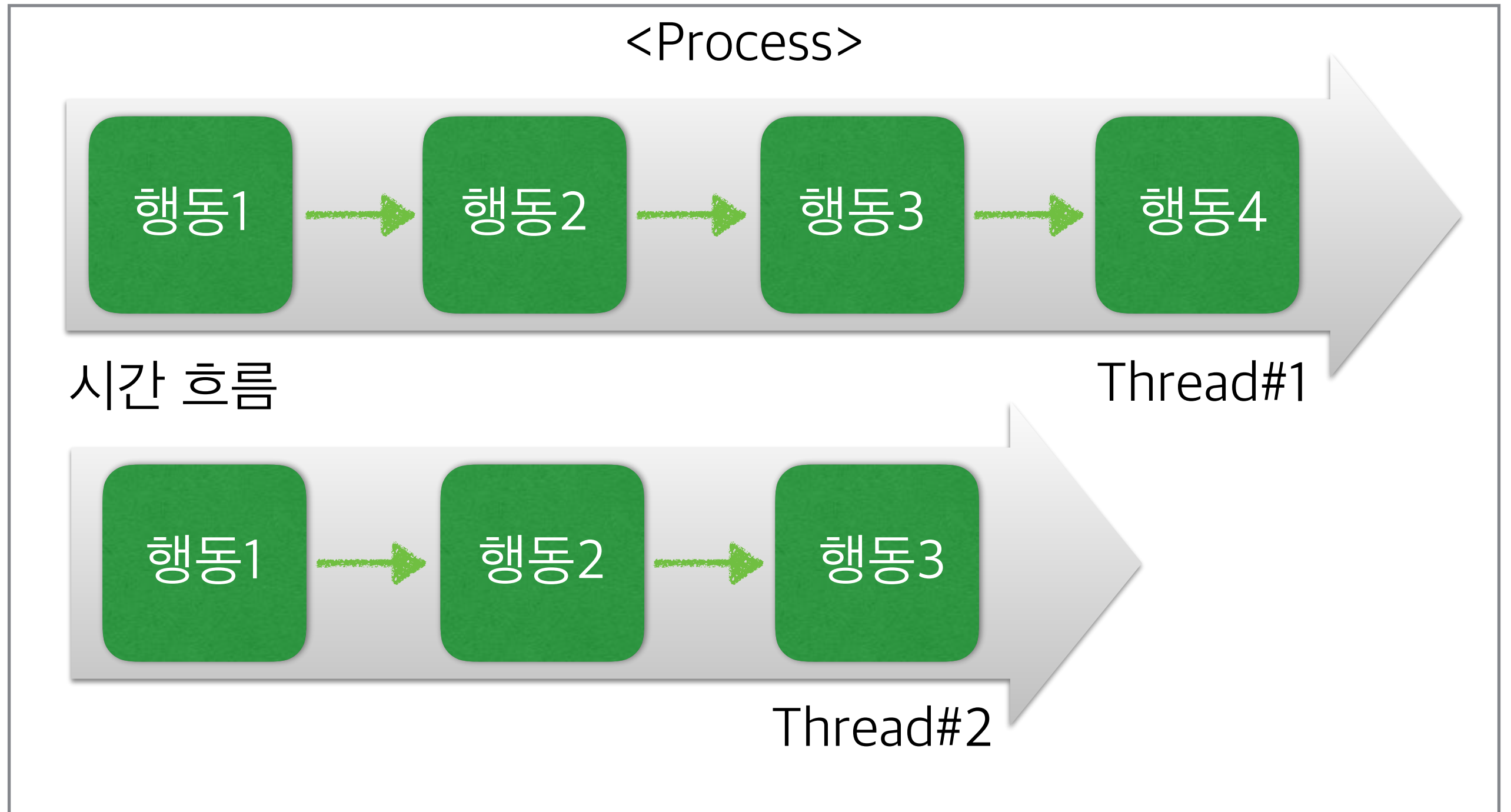
- 만약 작업시간이 오래걸리는 작업을 main thread에서 실행된다면 어떻게 될까??

# Single Thread

---



# Multi Thread



# When is use

---

- 동시에 작업이 필요한 경우
- Multi core process를 사용하기 위해
- 중요한 작업에 방해를 받지 않기 위해
- 상태를 계속 감시 해야 할 경우가 필요할때

# Multi thread 사용 예

---

- Network request/response
- time control
- image download/upload
- long time actions



# Multi thread 사용 예

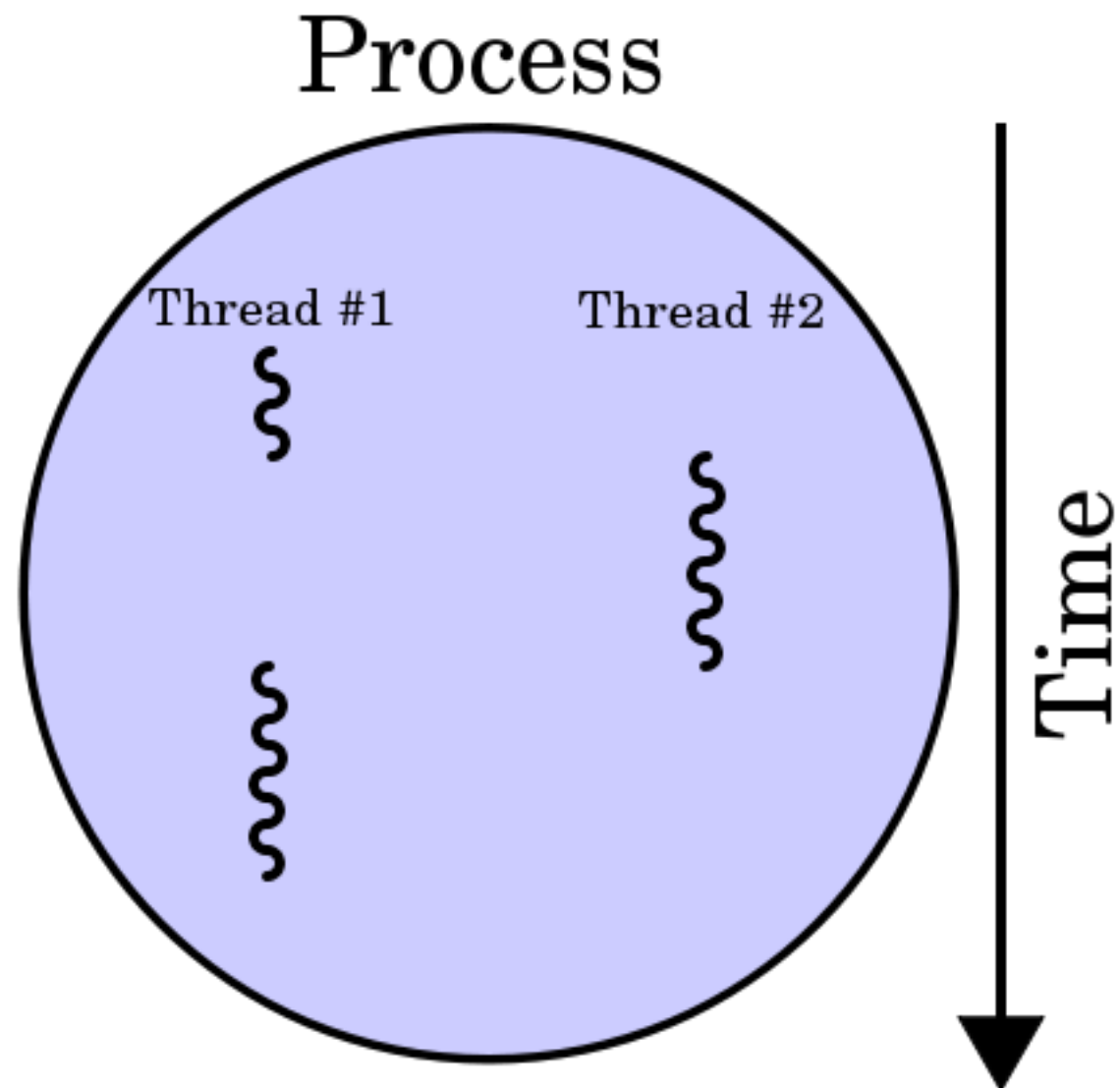
---

- 알고 있는 Multithread의 예에 대해서 말해보아요.

잠깐! 이런 용어는 알아야 쉽게 이해 가능합니다.

# Single Core Process

---



# 동기 방식 / 비동기 방식

---

- 동기 (synchronous) : 프로그래밍 적으로 어떤 일이 끝난후 다음 행동을 한다면 동기방식
- 비동기 (Asynchronous): 어떤 일이 끝난것과는 상관없이 다음 행동을 한다면 비동기 방식
- 디자인패턴에 의한 비동기처리는 다음과 같은 것들이 있다.
  - 델리게이트(delegate), 셀렉터(@selector), 블록(block), noti피케이션(Notification)

# 교착상태(deadlock)

---

- 교착 상태(膠着狀態, 영어: deadlock)란 두 개 이상의 작업이 서로 상대방의 작업이 끝나기만을 기다리고 있기 때문에 결과적으로 아무것도 완료되지 못하는 상태를 가리킨다

# 교착상태 추가 정보 읽어보기

---

- <http://goo.gl/jTXUXO>

---

# Multithread

---

강사 주영민

# iOS MultiThread방법

---

- Thread : 직접 thread를 만들어서 제어 하는 방식
- GCD : Closer기반의 기법으로 코드 가독성이 좋고 간편하다.
- Operation : GCD기반의 rapper Class. 간단하게 사용가능하고 고수준의 API를 제공한다. 성능이 느린편
- performSelector: Selector를 이용한 방식, ARC이전에 주로 사용한 방식이었으나 GCD이후엔 많이 사용되진 않는다.
- Timer : 간단한 interval Notification를 제공해 주는 Class. 특히 하게도 **Timer는 mainLoop**에서 실행된다.



---

# GCD(Grand Central Dispatch)

---

강사 주영민

# gcd

---

- 비동기로 여러작업을 수행시키는 강력하고 쉬운 방법이다.
- System에서 Thread관리를 알아서 해준다.
- dispatch queue를 이용해 작업들을 컨트롤 한다.
- work item : Closure를 활용해서 구현되어 있으며 queue를 생성할때 꼭 같이 만들어야 한다.

# DispatchQueue

---

- dispatch queue는 GCD의 핵심으로 GCD로 실행한 작업들을 관리하는 queue이다.
- 모든 dispatch queue는 first-in, first-out 데이터 구조이다.
- Serial Queue와 Concurrent Queue 2종류로 나눌수 있다.

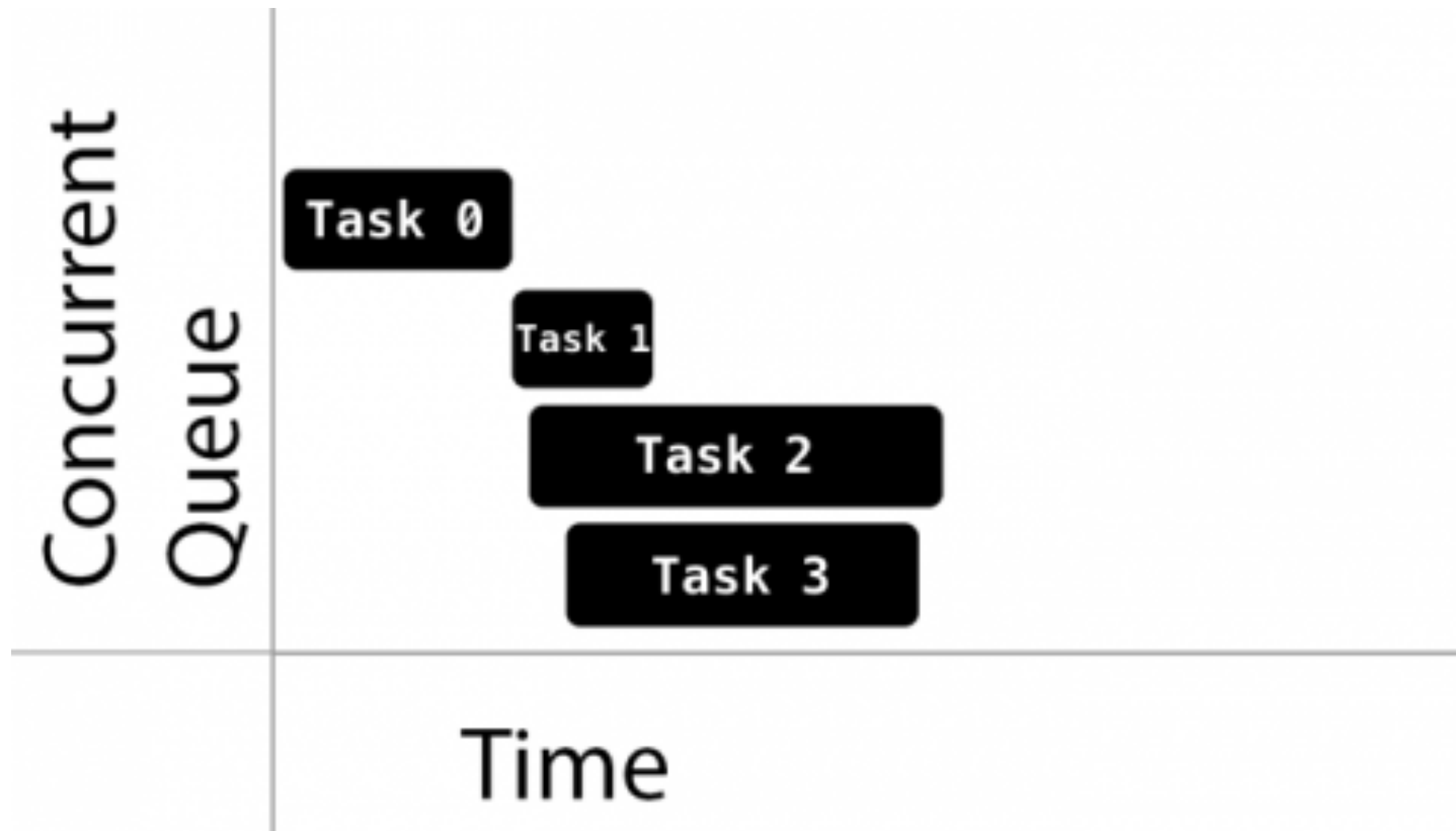
# Serial Queue

---



# Concurrent Queue

---



# init

---

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

```
let queue = DispatchQueue(label: "com.wing.queue")  
let queueOption = DispatchQueue(label: "com.wing.queue1", qos: .userInitiated)  
let queueAttri = DispatchQueue(label: "com.wing.queue2",  
                               attributes: [.concurrent, .initiallyInactive])
```

# Quality Of Service (QoS) and Priorities(우선순위)

---

- 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

# Attributes(속성)

---

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

- Default = 직렬
- concurrent = 병렬
- initiallyInactive = 태스크 수동 실행(activate() 메소드)



# async

---

```
let queue = DispatchQueue(label: "com.wing.async")
queue.async {
}
```

# Main dispatch queue

---

- Main Thread를 가르키며 기본 UI를 제어하는 queue이다.
- Serial 방식으로 들어온 순서대로 진행되며 앞에 작업이 종료될 때까지 뒤의 작업들은 대기 한다.
- 생성

```
DispatchQueue.main.async {  
    // Do something  
}
```

# Global dispatch queue

---

- app 전역에 사용되는 queue로서 Concurrent 방식의 queue이다.
- option으로 qos를 설정 할수 있다.
- 생성

```
let globalQueue = DispatchQueue.global()  
let globalQueue = DispatchQueue.global(qos: .userInitiated)
```

# DispatchWorkItem

---

- 실행 할수 있는 작업의 캡슐화
- 이벤트를 등록, 취소 할수 있다.
- dispatchQueue에서 실행 시킬수 있다.

```
public init(qos: DispatchQoS = default,  
            flags: DispatchWorkItemFlags = default,  
            block: @escaping @convention(block) () -> Swift.Void)
```

# DispatchWorkItem example

---

```
func useWorkItem() {  
    var value = 10  
  
    let workItem = DispatchWorkItem {  
        value += 5  
    }  
  
    workItem.perform()  
  
    let queue = DispatchQueue.global(qos: .utility)  
  
    queue.async(execute: workItem)  
  
    workItem.notify(queue: DispatchQueue.main) {  
        print("value = ", value)  
    }  
}
```

# Timer

---

```
let delayQueue = DispatchQueue(label: "com.wing.delayqueue", qos: .userInitiated)

print(Date())

let additionalTime: DispatchTimeInterval = .seconds(2)

delayQueue.asyncAfter(deadline: .now() + additionalTime) {
    print(Date())
}

delayQueue.asyncAfter(deadline: .now() + 0.75) {
    print(Date())
}
```