Concurrency

동시성

23.02.02 마차챠 세미나

비동기(async), 동시성(Concurrenct)의 개념

비동기(Asynchronous)

- 작업을 보내는 즉시 리턴을 하는 개념이 비동기이다.
- 일을 시작 시키고, 기다리지 않고 바로 다음 작업을 시작
- 작업을 처리하는 시간 자체가 더 오래 걸린다고 가정한다면, 작업이 끝나는 시간을 기다리지 않는것은 큰 의미가 있음

동기(Synchronous)

- 작업을 시키고, 다른 쓰레드에서 일이 끝날때까지 기다린다.
- 일이 다 끝나고 리턴
- 작업이 길어지면, 메인쓰레드에서만 일을 하는 것처럼 버벅이게 된다.

비동기(async), 동시성(Concurrenct)의 개념

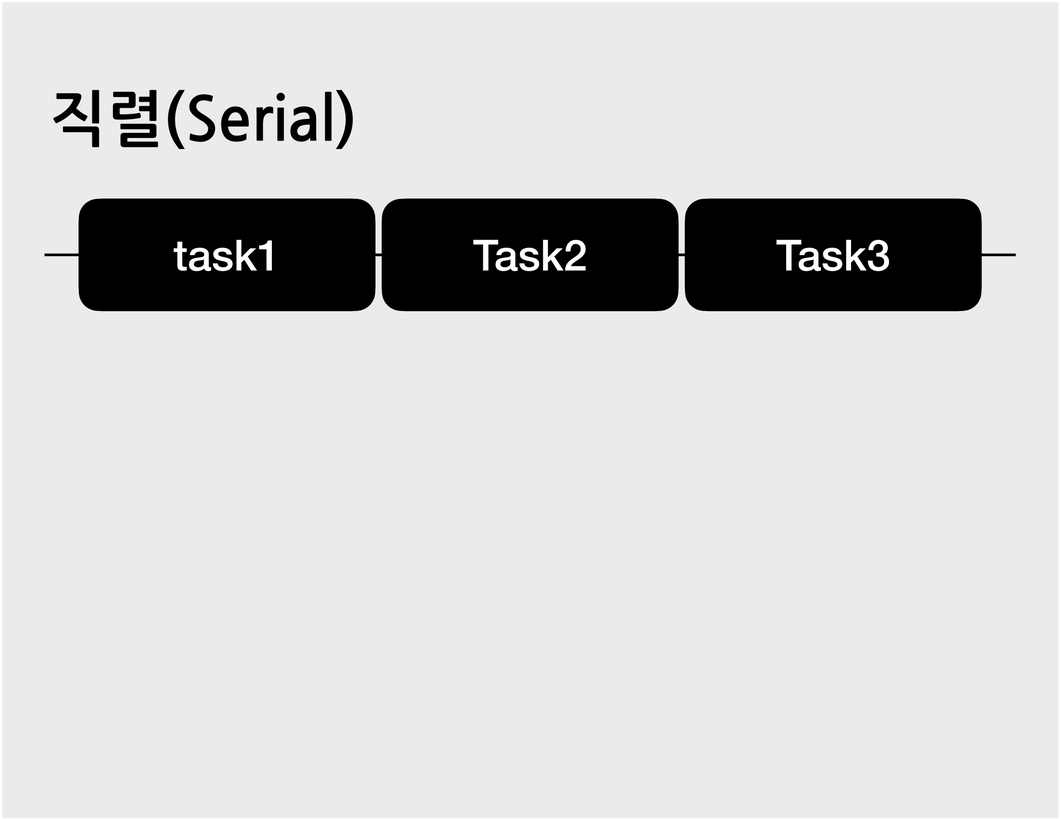
직렬(Serial)

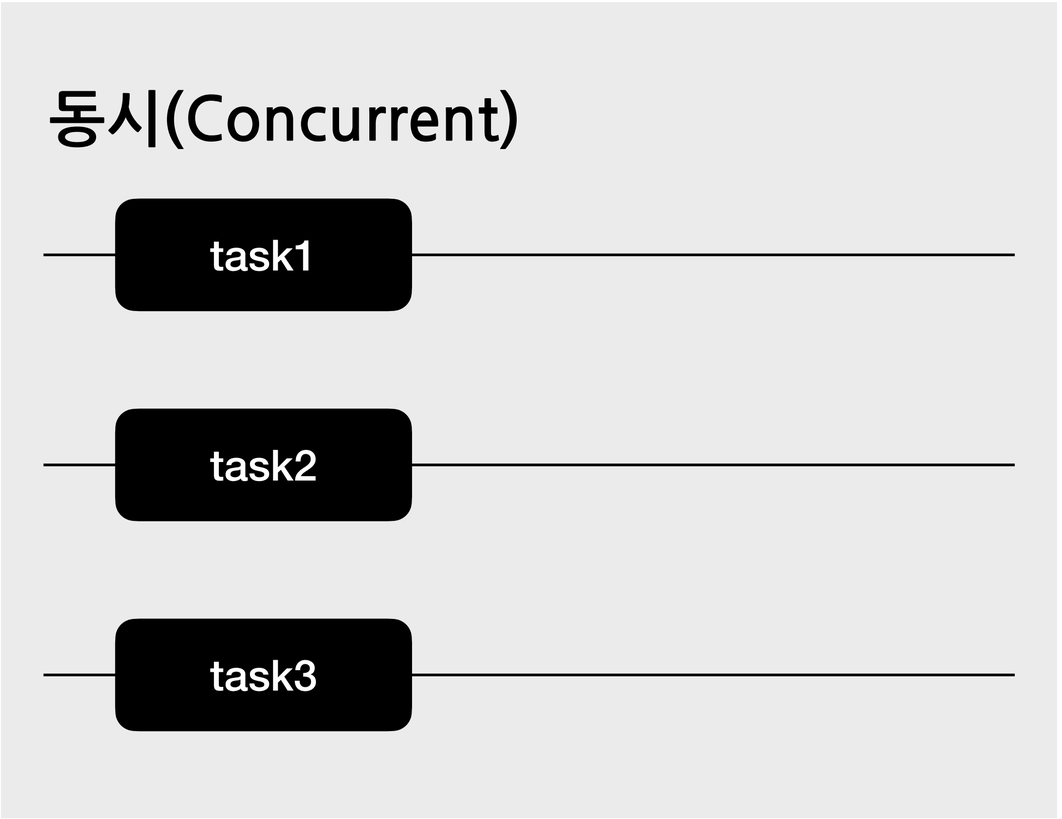
- 다른 한개의 쓰레드에서 순서적으로 일을 처리한다.
- 순서대로 일을 처리할 수 밖에 없다.

동시(Concurrent)

- 다른 여러개의 쓰레드에서 일을 하도록 분산처리가 된다.
- 독립적이지만 여러개의 유사한 작업을 빨리빨리 처리하고 싶을 때
- 여러개의 쓰레드를 사용해서 일처리를 할 수 있음

비동기(async), 동시성(Concurrenct)의 개념





비동기 처리가 필요한 이유

- 메인 쓰레드에서 네트워크 통신(서버에 데이터를 요청)과 같은 작업은 부하가 많이 걸리는 작업을 하게되면 앱이 버벅거리게 된다.
  - 메인 쓰레드는 Update Cycle마다 화면을 다시 그려야하기 때문에 부하가 많은 작업을 시키면 안된다.

- 다른 쓰레드로 분산처리(=동시성 프로그래밍)을 통해 이러한 문제를 해결할 수 있다.

Update Cycle 시간이 매우 짧아서 메인쓰레드에서 그 시간보다 오래걸리는 작업을 하면 화면이 버벅이게 되어 비동기 처리가 필요하다.

async/await의 도입 이유

비동기 함수를 이어서 처리할 때 코드를 들여쓰고, 다시 클로저를 이은 다음 들여써야한다.
비동기 함수의 일이 종료되는 시점을 연결하기 위해, 끊임없는 콜백함수의 연결과 들여쓰기 사용

```
func processImageData1(completionBlock: (_ result: Image) -> Void) {
    loadWebResource("dataprofile.txt") { dataResource in
        loadWebResource("imagedata.dat") { imageResource in
            decodeImage(dataResource, imageResource) { imageTmp in
                dewarpAndCleanupImage(imageTmp) { imageResult in
                    completionBlock(imageResult)
                }
            }
        }
    }
}
```

async/await의 도입 이유

위와 같은 단점을 극복하기 위해 async/await를 사용한다.

함수에서 async(비동기 함수)로 정의, 리턴 방식 사용
작업이 끝나는 시점을 await를 통해 기다리고 실행할 수 있게 한다.
그 결과 들여쓰기를 하지 않아도 되고, 코드가 어디서 멈추는지 알 수 있게 되어
코드처리가 깔끔해진다.

async/await의 도입 이유

```
func processImageData( ) async throws -> Image {
    let dataResource  = try await loadWebResource("dataprofile.txt")
    let imageResource = try await loadWebResource("imagedata.dat")
    let imageTmp      = try await decodeImage(dataResource, i..R..rce)
    let imageResult   = try await dewarpAndCleanupImage(imageTmp)
    return  imageResult
}
```

동시성 프로그래밍과 관련된 문제점

1) 경쟁상황 / 경쟁 조건 (Race Condition)

- 경쟁상황 / 경쟁조건 = Thread-safe하지 않음
  - 멀티 쓰레드의 환경에서, 같은 시점에 여러개의 쓰레드에서 하나의 메모리에 동시접근 하는 문제
- 같은 시점에 동시에 접근을 못함 → 메모리를 쓰고있는 동안에는 여러 쓰레드에서. 접근 못하도록 Lock(잠금) → Thread-Safe 처리

2)교착상태 (Deadlocks)

- 멀티 쓰레드 환경에서, 배타적인 메모리사용으로 일이 진행이 안되는 문제
- 2개 이상의 쓰레드가 서로 잠그고 점유하려고 하면서 메서드의 작업이 종료도 못하고 일의 진행이 멈춰버리는 상태