Day 10 - 동시성과 Coroutine

오늘 배울 것

- 배경 지식
- 코루틴 (Coroutine)

배경 지식

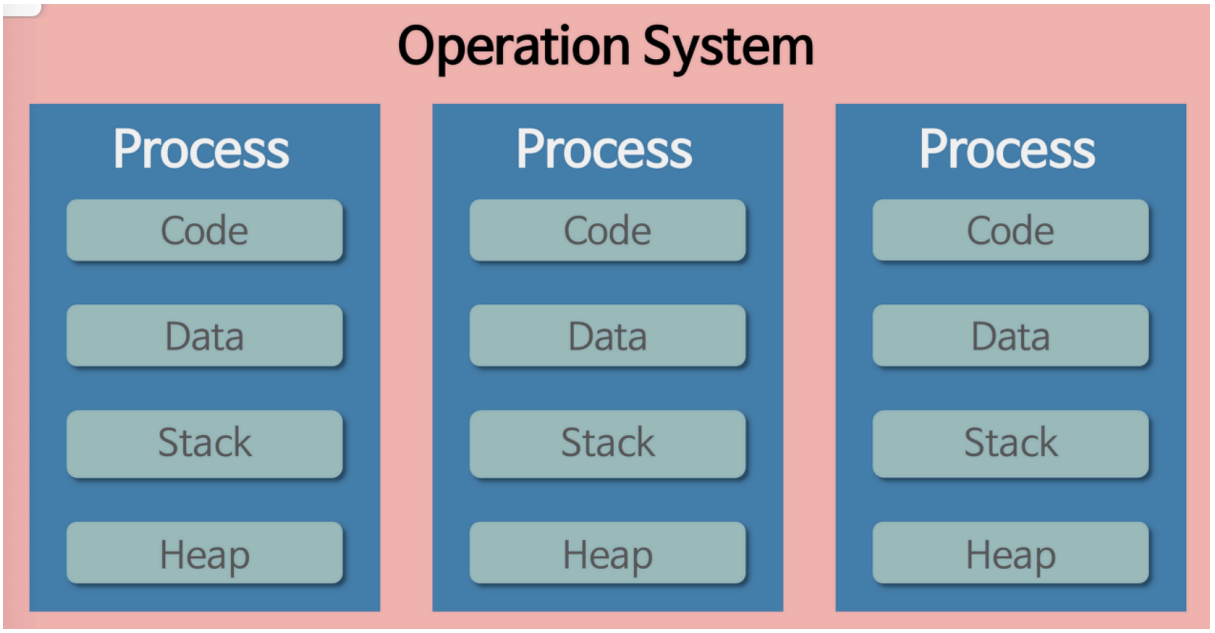
많은 사람들이 헷갈려 하는 것에 대해 먼저 짚고 넘어갑니다.

기본적인 CS 지식이기도 합니다.

프로세스와 스레드(Thread)

프로세스와 스레드의 차이에 대해 간략히 짚고 넘어가겠습니다.

프로세스

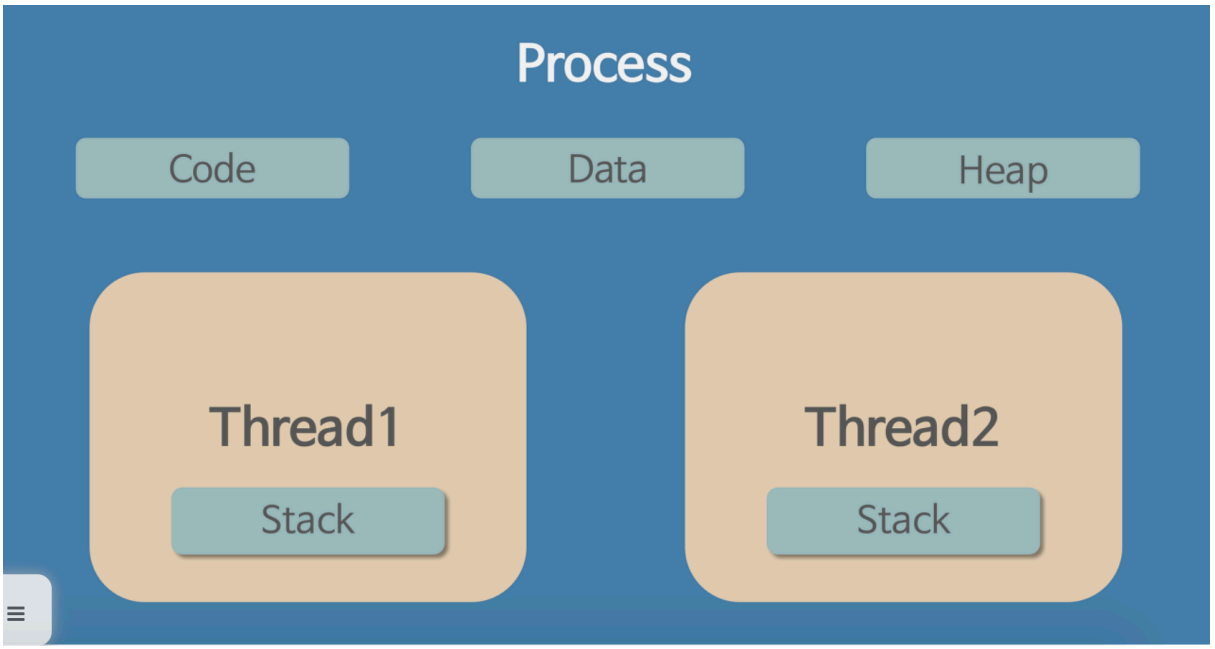


프로그램이 실행되면, **메모리**를 할당 받게 됩니다. 메모리는 크게 4가지 영역으로 이루어져 있습니다.

1. **코드 영역**
    - 프로그래머가 작성한 함수의 코드가 CPU가 해석 가능한 기계어 형태로 저장되어 있습니다
2. **데이터 영역**
    - 코드가 사용하는 전역 변수나 각종 데이터들이 모여있습니다
3. **스택 영역**
    - 지역 변수와 같은 호출한 함수가 종료되면 되돌아올 임시 자료를 저장하는 독립 공간
    - 함수를 호출할 때, 함수를 호출한 주체의 각종 값이 이곳에 저장된다고 이해해도 됨
4. **힙 영역**
    - 동적으로 할당되는 데이터들을 위해 존재하는 공간

중요한 점은, 각 프로세스는 별도의 공간에서 실행되고, **프로세스 끼리는 자원 공유를 하지 않는다는 점**입니다. 따라서 별도의 통신이 필요합니다. (IPC, 파이프, 소켓 등)

스레드 (Thread)



스레드는 **프로세스 내부의 실행 단위**입니다. **Stack을 따로 할당** 받습니다. **Code, Data, Heap 영역은 공유합니다.**

멀티스레드는 특정 자원을 공유하여 사용하기 때문에 시스템의 자원과 처리 비용이 멀티프로세싱에 비해 어느 정도 감소할 수 있습니다. 이는 통신의 부담이 적기 때문입니다.

따라서 한 시스템 안에서 특정 일에 대한 **동시성 작업**이 필요한 경우, 일반적으로 멀티프로세싱 보다는 멀티스레딩을 선호합니다. 일반적으로 멀티스레딩 구현이 된 프로그램의 경우, CPU 또한 병렬적으로 사용하므로, 처리 시간이 빠르다는 장점이 있습니다.

그렇다면 파이썬도 그럴까요? 그렇지 않습니다.

GIL 때문입니다.

GIL (Global Interpreter Lock)

파이썬은 일반적으로 메인 쓰레드가 코드를 차례차례 실행합니다.

아래 코드를 실행해봅시다

```
import time
import threading
import random

list_len = 10_000_000

def working():
    return sorted([random.random() for _ in range(list_len)])

start_time = time.time()
sorted_list1 = working()
print(f"first working ⟶ {time.time() - start_time:.4f}s")

start_time = time.time()
sorted_list2 = working()
print(f"second working ⟶ {time.time() - start_time:.4f}s"
```

이런 결과가 나왔습니다

```
first working —> 3.4968s
second working —> 3.3992s

# TOTAL ~= 6.8s
```

파이썬에서 멀티 쓰레딩을 위해선 `threading` 모듈이나 `thread` 모듈을 사용할 수 있습니다. (고수준, 저수준 구현 라이브러리라는 차이가 있으며, `threading`에서 `thread` 모듈을 사용 중입니다)

해당 라이브러리의 사용법은 간단합니다.

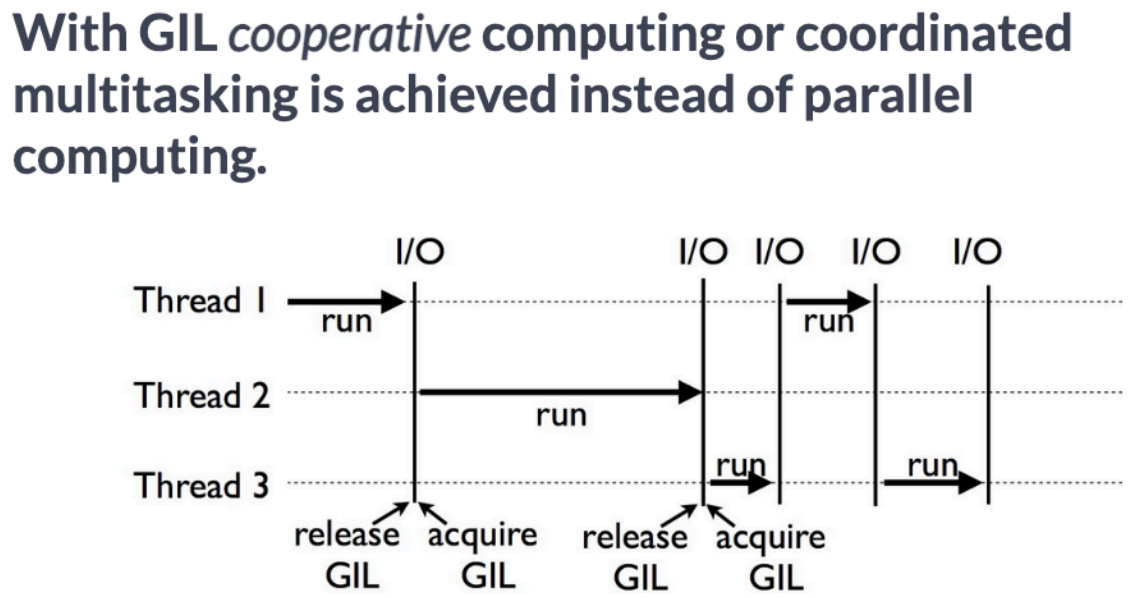
1. `threading.Thread()` 함수를 호출해 객체를 얻습니다.
2. Thread 객체의 `start()` 함수를 호출합니다

이 방법으로 위 코드를 다시 구현해보겠습니다.

```
def working():
    return sorted([random.random() for _ in range(list_len)])

threads = []
start_time = time.time()
for i in range(2):
    threads.append(threading.Thread(target=working))
    threads[i].start()

 for t in threads:
    t.join()        # join 메소드 사용시, thread 수행될 때까지 기다림

print(f"multi-threading —> {time.time() - start_time:.4f}s") # 6.9964s
```

동작 시간 이 줄지 않았습니다. 왜 일까요?

파이썬 인터프린터 구현체 안에, 멀티 스레딩을 막는 **Global Interpreter Lock**이 있기 때문입니다.



위 그림에서는 몇가지 정보를 알 수 있습니다

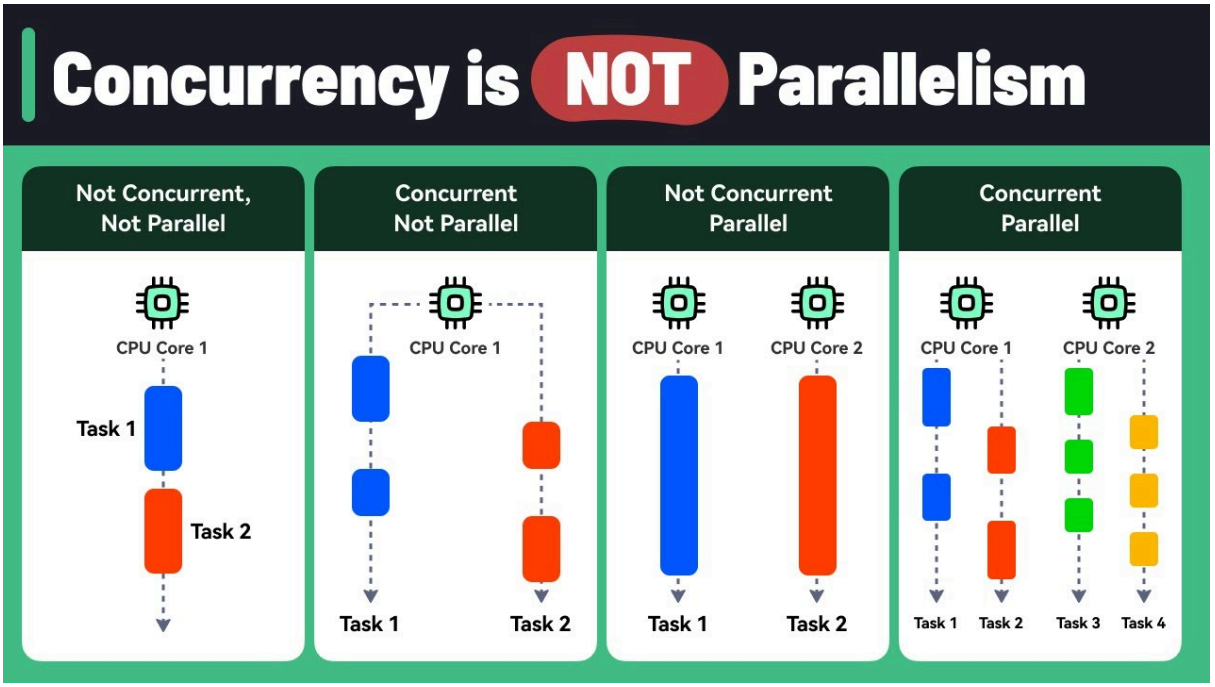
- 멀티 스레딩이 불가능 한 것은 아니다
- 그러나 **한 번에 하나의 스레드**만 실행된다.
- **I/O**가 일어나는 시점에 스레드 락이 해제된다. (읽기/쓰기가 많은 작업에는 이점이 있겠죠?)

이는 파이썬이 메모리를 관리하는 특성 때문입니다. 한 번에 여러 스레드가 특정 메모리에 접근해 race condition이 발생하는 것을 막기 위해서 Lock을 걸어둔 것입니다. (더 궁금하면 **python reference counter**를 찾아보세요)

따라서 파이썬에서 병렬성/동시성 작업을 수행하기 위해서는 **멀티 프로세싱**을 사용해야 하며, 이는 파이썬 웹 서버 또한 자유롭지 않은 문제입니다.

(예를 들어, FastAPI가 동시성 웹 프레임워크로 사랑 받고 있지만, 실제로 큰 부하를 견디지는 못합니다)

병렬성(Parallelism) vs 동시성 (Concurrency)



위 설명에서 병렬성/동시성이라는 용어를 같이 사용했는데요. 이번에는 이 차이에 대해 알아봅시다.

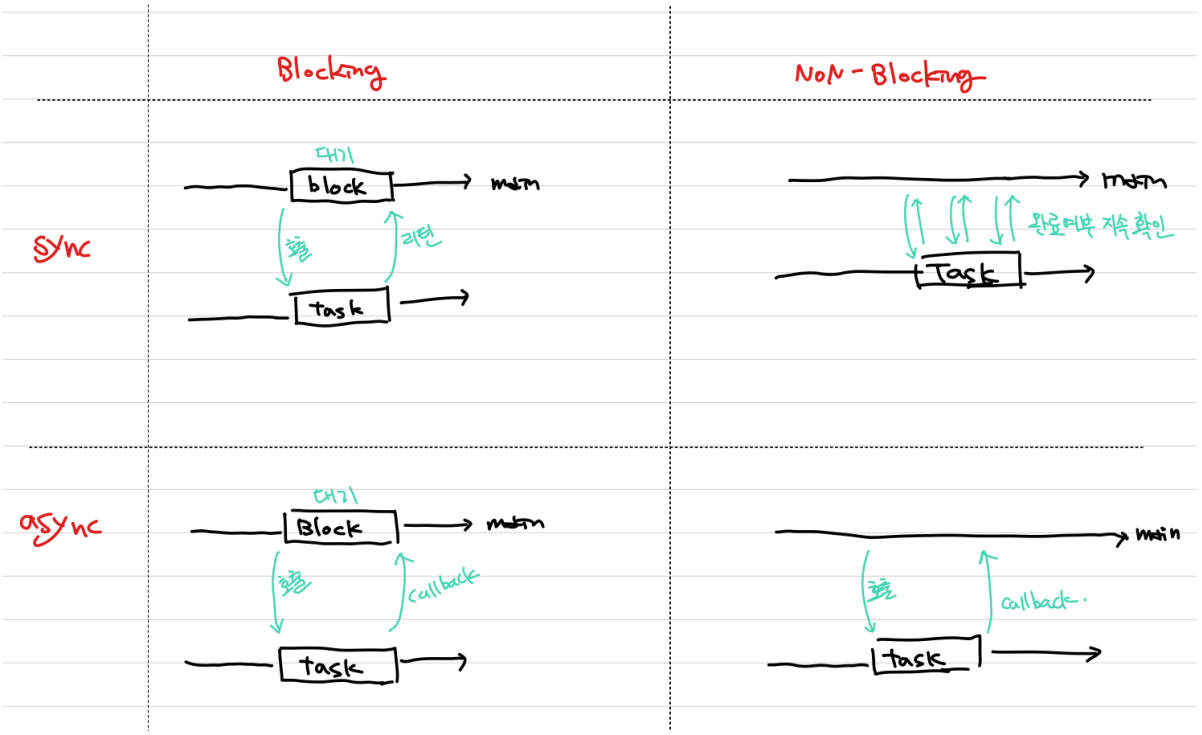
위 그림을 봅시다. 위 그림은 **병렬성**과 **동시성**의 차이에 대해 묘사하고 있습니다.

생각보다 많은 사람들이 이 두 개념을 혼용하곤 합니다. 또한 이름만 들었을 때, 이 둘의 차이를 알기 어렵기도 합니다.

들어가기 앞서, 개념을 명확히 하고 넘어갑시다. 두 개념의 차이를 표로 나타내면 아래와 같습니다.

동시성 (Concurrency)	병렬성 (Paallelism)
동시에 실행되는 것 처럼 보이는 것	실제로 동시에 실행되는 것
논리적인 개념	물리적인 개념
싱글코어, 멀티코어 모두에서 가능	멀티코어에서만 가능

Async vs Non-Blocking



또 다르게 혼용 되는 개념으로는, **Blocking/Non-Blocking**과 **동기(sync)/비동기** 가 있습니다.

두 개념은 비슷한 것을 설명하는 것 같지만, 서로 다른 차원의 작업을 수행하는 방식을 설명합니다.

간략히 설명하고 넘어가자면, 아래와 같습니다.

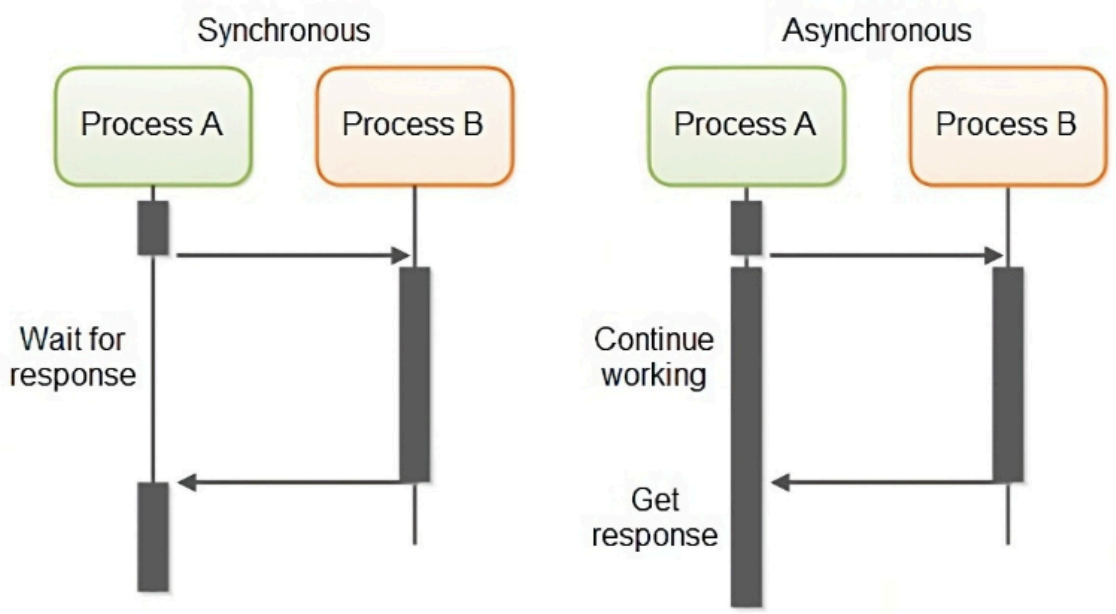
- **동기/비동기**는 작업을 순차적으로 수행할지 아닐지에 대한 개념입니다.
- **블로킹/논블로킹**은 현재 작업이 block(차단) 되느냐 아니냐에 대한 개념입니다.

두 개념의 헷갈리는 부분을 알아보기 전, 먼저 두 개념을 알아봅시다.

Synchronous (동기) 와 Asynchronous (비동기식)

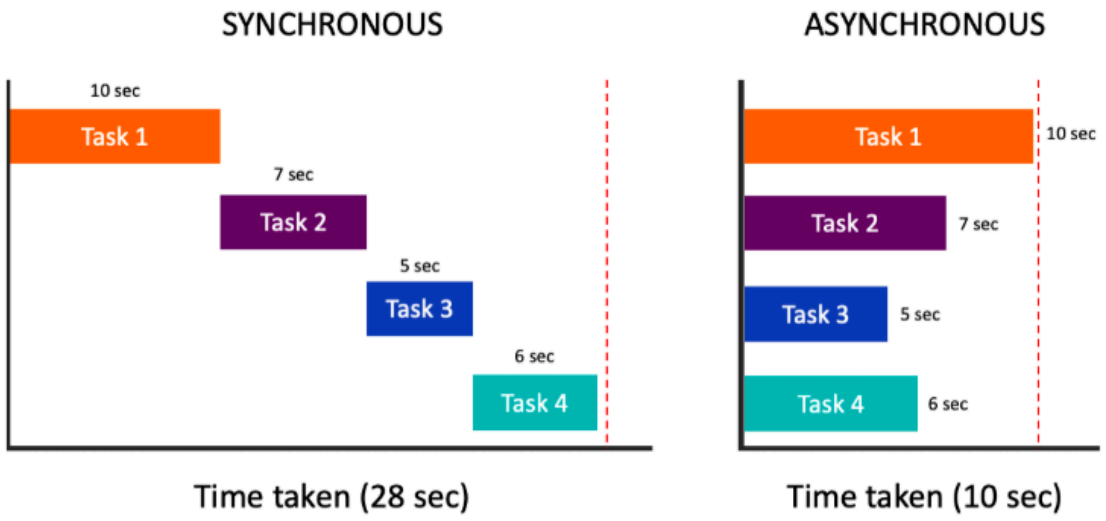
chrono는 그리스어로 시간입니다. sync는 그리스어로 함께라는 뜻입니다.

이 단어의 유래를 생각하면 조금 더 이해하기 편합니다.



동기는 작업 B가 완료되어야 다음 작업을 수행하고, 비동기는 작업 B의 완료 여부를 따지지 않고 바로 다음 작업을 수행한다

- Synchronous는 작업을 시간에 맞춰 함께 실행한다는 것입니다.
  - 즉, 요청한 작업에 대한 완료 여부를 따져, 순차처리 한다는 것입니다.
- Asynchronous는 Synchronous의 부정형으로, 요청한 작업에 대해 완료 여부를 따지지 않는다는 것입니다.
  - 즉, 자신의 다음 작업을 이어서 수행한다는 것입니다.



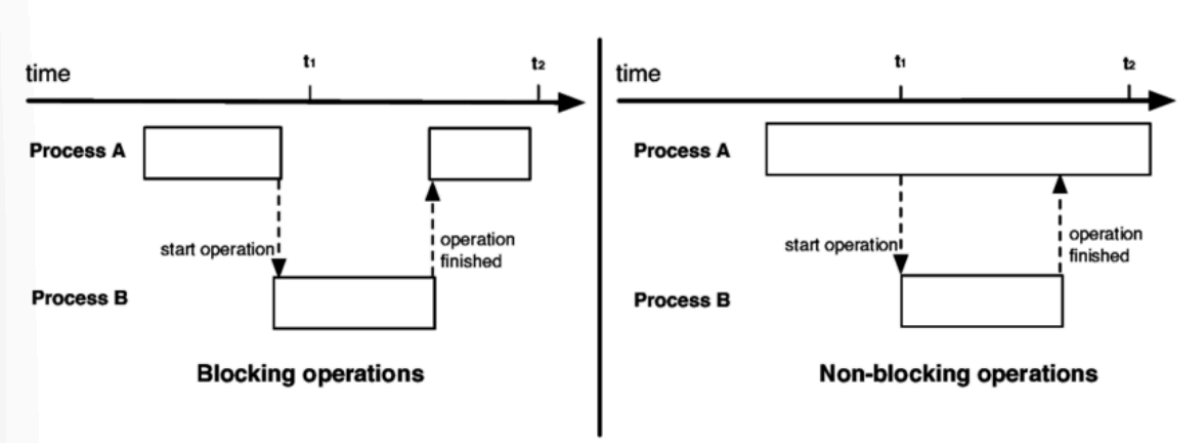
일반적으로 비동기로 처리하는 경우, 경우, I/O에 대한 이점을 얻게 됩니다.

프로그래밍에서 가장 비싼 것은 디스크와 네트워크입니다.

무언가를 디스크나 네트워크로 읽거나 쓸 때, 가장 많은 시간을 기다리게 됩니다. 비동기로 처리하면 전반적인 시스템 향상에 도움을 줍니다.

다만, 비동기 처리 시에는 작업 요청에 대한 처리 순서가 보장되지 않으니, 이 점을 주의해야 합니다.

Blocking / Non-Blocking



Sync / Async가 작업 순서에 대한 작업에 대한 순차 처리에 대한 차이를 이야기 하는 것이라면, blocking과 non-blocking은 다른 주체의 작업과 관련 없이 자신의 작업을 하는 것에 대한 차이를 다루고 있습니다.

- 내가 호출한 작업에 대해 기다려야 한다면 Blocking
- 내가 호출한 작업에 대해 기다리지 않는다면 Non-Blocking입니다

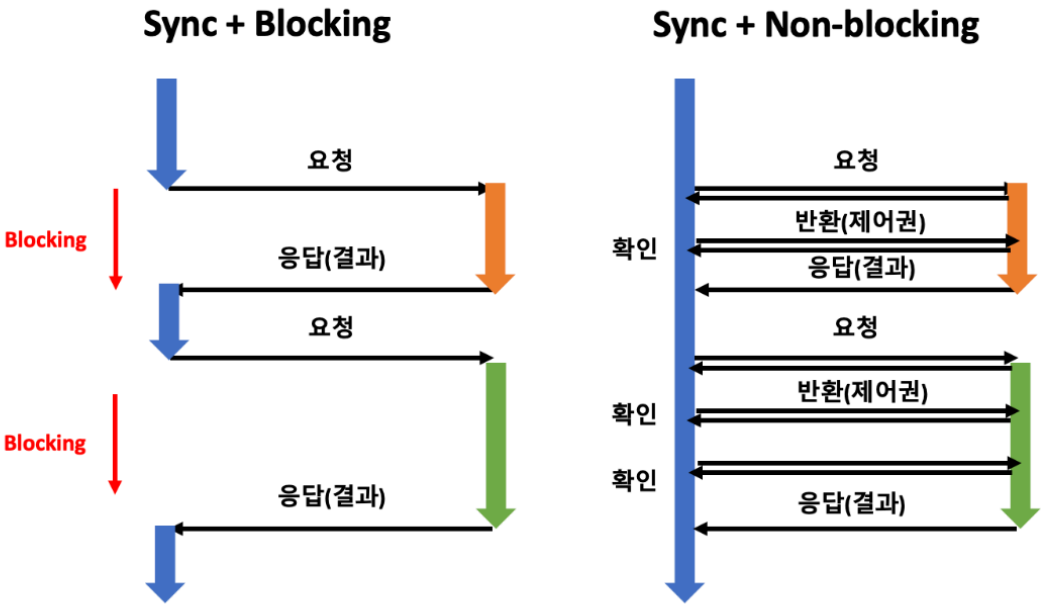
이를 CPU 제어권에 대한 차이로 보기도 합니다.

- 나를 호출한 주체에 CPU 제어권을 주지 않는다면 Blocking
- 나를 호출한 주체에게도 제어권을 주어 다른 일을 할 수 있게 한다면 Non-Blocking입니다.

Sync/Async & Blocking/Non-Blocking 조합

자, 그럼 이제 이런 개념들이 어떻게 조합되는지 보겠습니다.

대부분의 Async가 Async + Non-Blocking으로 동작하기 때문에 오해하기 쉬운 측면이 있습니다.



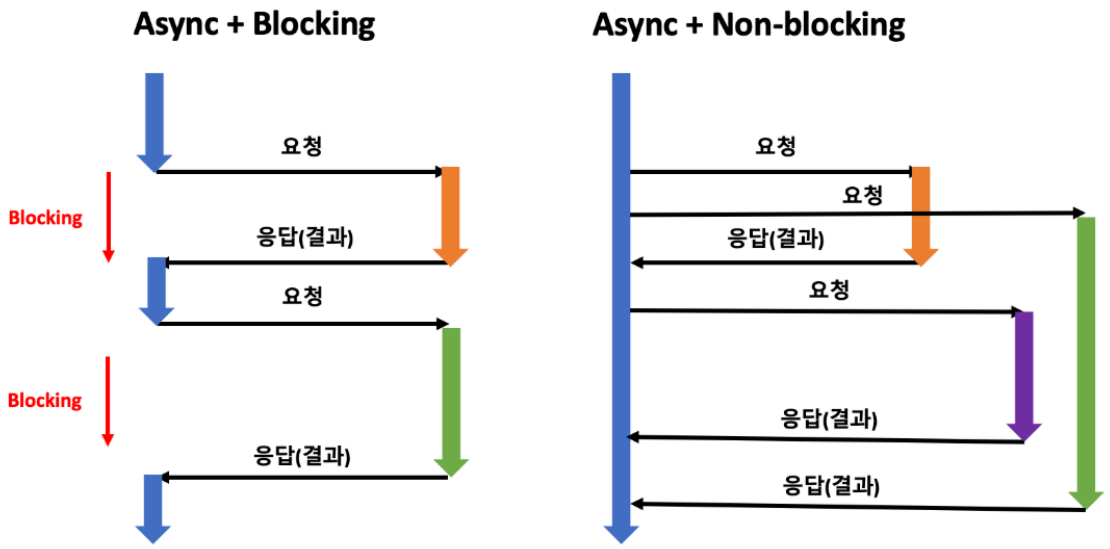
1. **Sync + Blocking**

A가 B를 요청하면, B는 A를 Blocking하고 실행됩니다. B가 끝나면 요청을 반환하고, A가 이어 실행됩니다.

2. Sync + Non-Blocking

이전과 다르게 B와 A는 동시에 동작합니다. 다만 A는 계속해서 B의 작업이 끝났는지 확인하는 작업을 거치게 됩니다. 이 작업을 **Polling** 이라고 합니다.

(시간이 남으면 spin lock에 대해 알아보세요)



3. **Async + Blocking**

크게 의미 없는 부분입니다. 순서는 상관 없지만, 응답이 올때 까지 기다려야 합니다. 사실 Sync + Blocking과 크게 다른 점도 없습니다. (의도치 않게 구현되는 모델입니다)

4. Async + Blocking

일반적으로 자원과 시간이 효율적인 구조입니다. 자신의 일을 하면서 여러 작업을 요청 할 수 있습니다. (자바 스크립트에서 Callback 많이 사용하죠?)

제너레이터와 코루틴 (Coroutine)

파이썬에서 동시성(Concurrency) 프로그래밍을 어떻게 지원하는지 살펴보겠습니다. (**병렬성이 아니라는 점을 명심하세요**)

코루틴이란 용어는 파이썬만의 용어는 아닙니다. 함께를 뜻하는 **Co-**와 루틴(routine)의 합성어로, 여러 루틴이 협력적으로 실행을 제어하는 프로그래밍 패턴을 이야기합니다.

일반적으로 가장 먼저 보게 되는 비동기 프로그래밍관련 문법은, genereator와 coroutine 입니다. 두 문법이 서로 유사하기도 한 만큼 이 부분을 먼저 알아 보겠습니다.

제너레이터

제너레이터는 값을 하나씩 반환하는 함수입니다.

yield 키워드를 사용해 값을 반환하고, 다시 호출되면 일시 중단된 지점에서 실행을 다시 시작합니다.

이를 두고 **느슨하게 평가된다** 라고 하기도 합니다.

아래 코드를 실행해봅시다.

```
def gen_numbers(n):
    i = 1
    while i <= n:
        yield i
        i += 1

for i in gen_numbers(5):
    print(i)
```

아래와 같은 출력이 나옵니다

```
1
2
3
4
5
```

아래 코드를 봅시다.

```
def read_large_file(file_path):
    """Generator that reads a large file line by line."""
```

```
    with open(file_path, "r") as file:
        for line in file:
            yield line  # Yield one line at a time

# Example Usage
file_path = "large_text_file.txt"  # Assume this file is very large
for line in read_large_file(file_path):
    process(line)  # Replace with actual processing logic
```

위 경우에서, generator를 사용함으로 한번에 많은 메모리를 사용하지 않을 수 있었습니다. 이런 경우 제너레이터를 사용함으로 이점을 얻을 수 있습니다.

코루틴 사용법

- 내부에 **yield** 키워드가 있는 함수입니다.
- 호출자가 실행을 컨트롤하며, 내부에 데이터를 전달할 수 있습니다.
    - send()를 호출하면 코루틴이 **호출자로부터 데이터를 받을 수 있습니다**
- 변수[외부에서 받음] = (yield 변수[외부 전달])
- 변수 = next(코루틴 객체)

```
def coroutine_processor():
    """데이터를 받아 평균을 계산하는 코루틴"""
    total = 0
    count = 0
    try:
        while True:
            data = yield  # 데이터를 받음
            if data is None:  # 종료 신호
                print("코루틴 종료.")
                yield  # StopIteration 방지
                break
            total += data
            count += 1
```

```
            print(f"받은 값: {data}, 현재 평균: {total / count}")
    except GeneratorExit:
        print("코루틴이 종료되었습니다.")

# 사용 예시
coro = coroutine_processor()
next(coro)  # 코루틴 초기화

for value in [10, 20, 30, 40, 50]:
    coro.send(value)

coro.send(None)  # 종료 신호 전송
```

다음 시간에는 asyncio를 사용한 비동기 ㅇ프로그래밍에 대해 알아보겠습니다.