파이썬 스터디

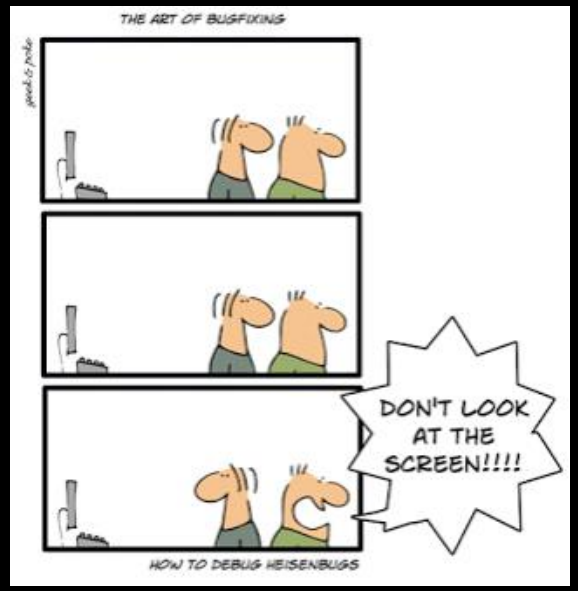
11주차

Ch40. 많은 함수를 동시에 실행하려면 코루틴을 고려하자

스레드를 사용한 동시성은 3가지 문제를 가지고 있습니다.

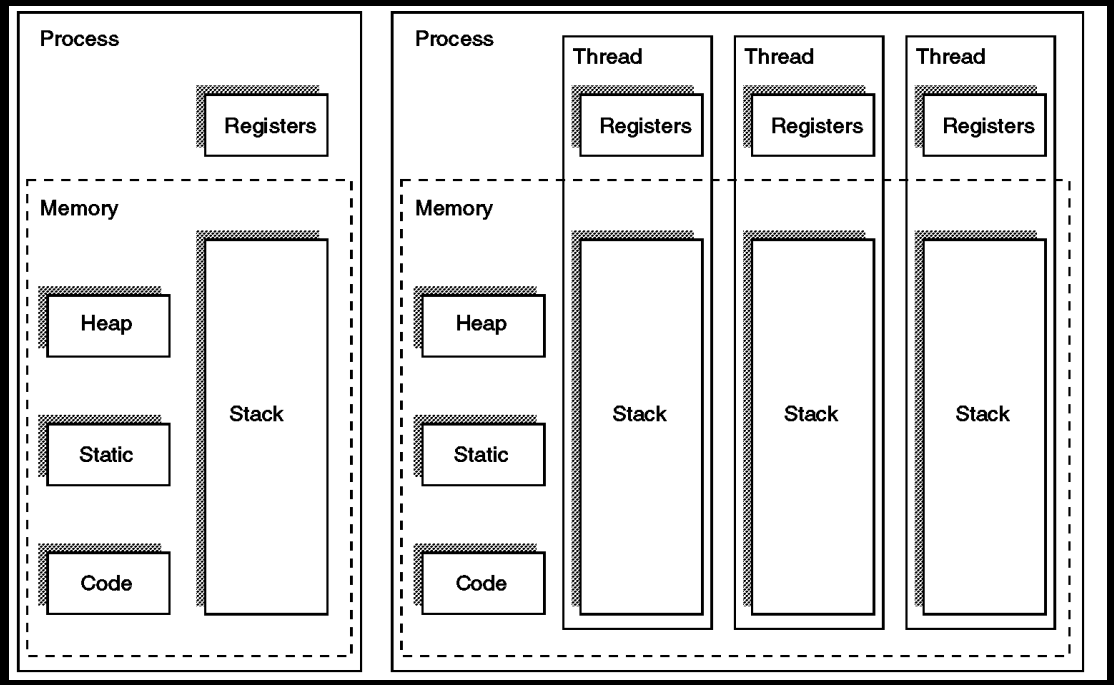
1. 순차 일관성(Sequential consistency)을 보장하는데 특별한 도구(Lock, Condition variable etc...)가 필요하며 싱글 스레드 코드보다 이해하기 어렵습니다.

   - Data race
   - Heisenbug



2. 스레드는 메모리를 많이(스레드당 8MB 정도) 소모 합니다.

- 윈도우 스레드의 경우에는 스택을 위한 메모리 1MB 필요
- Context를 위한 메모리가 필수적으로 필요



3. 스레드를 시작하는데 비용이 많이 듭니다.

- 스레드 스택을 위한 메모리 할당
- 컨텍스트 생성을 위한 비용

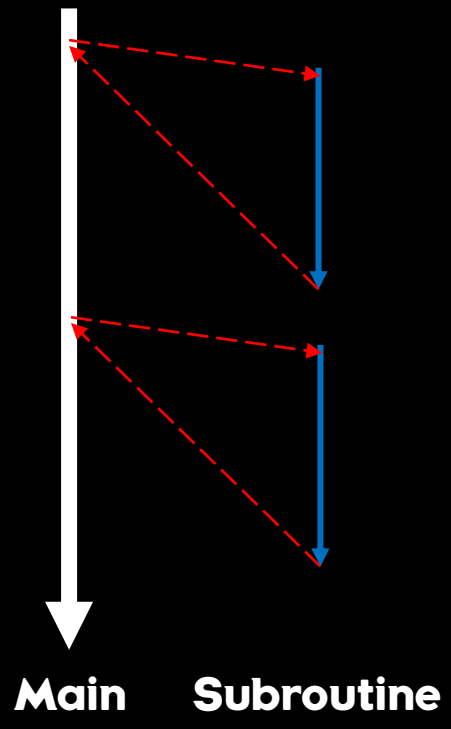
파이썬에서는 코루틴으로 이런 문제를 모두 해결합니다.

코루틴(Coroutine)은 뭘 까요?

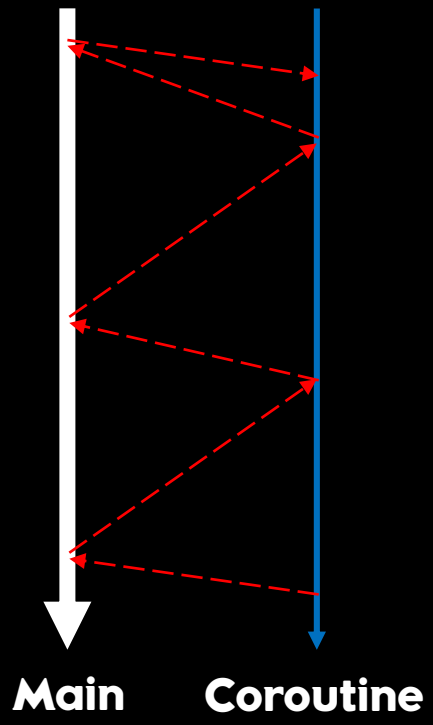
일단 서브 루틴(Subroutine)에 대해서 집고 넘어가려고 합니다.

서브 루틴은 특정 목적을 지닌 작업을 처리하는 코드의 모음 입니다.

메인 루틴에서 **호출되어 리턴 할 때 까지**를 하나의 처리 단위로 합니다.



반면 코루틴은 일련을 코드 수행을 중단한 다음 메인 루틴으로 돌아간 다음 다시 중단한 부분 부터 수행을 재개할 수 있습니다.



제너레이터와 다른 점은 send함수를 통해서 제너레이터 함수에 값을 전달할 수 있다는 점입니다.

```
def coroutine():
    while True:
        received = yield
        print('Received:', received)

it = coroutine()
next(it)            # 코루틴 준비
it.send('First')
it.send('Second')
>>>
Received: First
Received: Second
```

Send로 값을 전달받으면서 외부로 값을 전달하는 코루틴은 다음과 같이 작성할 수 있습니다.

```
def minimize():
    current = yield
    while True:
        new = yield current
        current = min(new, current)

it = minimize()
next(it)
print(it.send(100))        #100
print(it.send(50))         #50
print(it.send(60))         #50
print(it.send(40))         #40
print(it.send(12))         #12
```

그럼… 코루틴이 스레드의 문제를 해결 할 수 있을까요?

1. 코루틴은 **확실히 순차 일관성을 보장**할 것입니다.
   - 멀티 코어를 사용하는게 아니기 때문에…
   - 코드의 이해도는 코루틴 함수의 복잡도에 따라 다를 것 같습니다.

스레드를 사용한 동시성은 3가지 문제를 가지고 있습니다.

1. 순차 일관성(Sequential consistency)을 보장하는데 특별한 도구(Lock, Condition variable etc…)가 필요하며 싱글 스레드 코드보다 이해하기 어렵습니다.

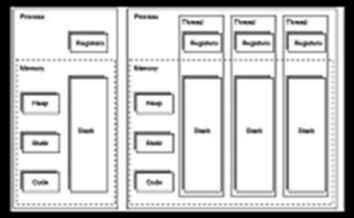
   - Data race
   - Heisenbug

2. 코루틴은 스레드 보다 적은 메모리를 사용할까요?

- 코루틴도 자신의 스택과 context 정보는 가져야 할 필요가 있습니다.
- Boost의 경우 스택 없는 코루틴이 있으나 최상위 루틴에서 일시정지 되는 것만 가능한 제약이 있습니다.

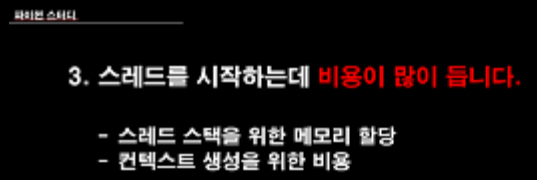
2. 스레드는 메모리를 많이(스레드당 8MB 정도) 소모 합니다.

- 윈도우 스레드의 경우에는 스택을 위한 메모리 1MB 필요
- Context를 위한 메모리가 필수적으로 필요



3. 코루틴은 스레드 보다 비용이 적을 것입니다.

- 결국은 함수의 호출이기 때문에 모든 레지스터를 저장하지 않아도 될 것입니다.



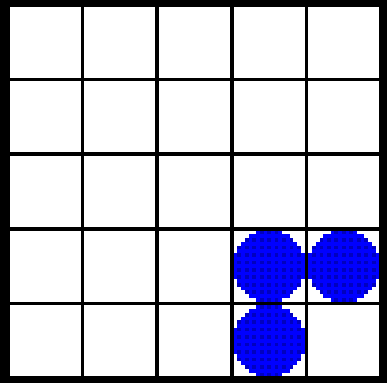
파이썬 구현체의 상세 내용은 잘 모르겠지만
저자가 언급하는 코루틴의 장점은 어느정도 납득은 갑니다.

이제부터 코루틴을 사용하는 예제를 살펴보겠습니다.

생명게임

존 호튼 콘웨이가 고안한 **Cellular Automata** 게임입니다.

정사각형의 여러 칸으로 나뉜 격자에서 각 칸마다 하나씩 존재하는 세포들이 **특정 규칙에 따라서 죽은 상태와 산 상태를 전환**하는 것을 살펴보는 게임입니다.



생명게임은 다음과 같이 진행됩니다.

1. 주변 격자의 상태를 취합.
2. 특정 규칙에 따라 자신의 상태를 변경
3. 1, 2의 과정을 전체 격자에 반복
4. 1, 2, 3의 과정을 여러 번 반복

1. 주변 격자의 상태를 취합.

```
def count_neighbors(y, x):
    n_ = yield Query(y + 1, x + 0) # 호출한 루틴에 좌표 값을 가진 객체를 반환하고
                                   해당 좌표 값의 상태를 입력을 받을 것을 기대합니다.
    Ne = yield Query(y + 1, x + 1)
    e_ = yield Query(y + 0, x + 1)
    se = yield Query(y – 1, x + 1)
    s_ = yield Query(y – 1, x + 0)
    sw = yield Query(y – 1, x – 1)
    w_ = yield Query(y + 0, x – 1)
    nw = yield Query(y + 1, x – 1)
    neighbor_states = [n_, ne, e_, se, s_, sw, w_, nw]
    count = 0
    for state in neighbor_states:
        if state == ALIVE:
            count += 1
    return count # ALIVE 상태인 격자의 수를 세서 반환합니다.
```

2. 특정 규칙에 따라 자신의 상태를 변경

```
def game_logic(state, neighbors):
    if state == ALIVE:
        if neighbors < 2:
            return EMPTY
        elif neighbors > 3:
            return EMPTY
    else:
        if neighbors == 3:
            return ALIVE
    return state
```

3. 1, 2의 과정을 전체 격자에 반복

```
def step_cell(y, x):
    state = yield Query(y, x) # 현재 상태를 입력을 받을 것을 기대합니다
    neighbors = yield from count_neighbors(y, x)
    next_state = game_logic(state, neighbors)
    yield Transition(y, x, next_state)
```

yield from은 코루틴을 조합하여 복잡한 코루틴을 구축하는데 사용됩니다. count_neighbors가 모두 소진되면 최종 결과 값이 neighbors에 담기게 됩니다.

3. 1, 2의 과정을 전체 격자에 반복

```
def simulate(height, width):
    while True:
        for y in range(height):
            for x in range(width):
                yield from step_cell(y, x)
        yield TICK # 전체 격자에 대해 반복이 끝났음을 나타내는 객체
```

4. 1, 2, 3의 과정을 여러 번 반복

```
def live_a_generation(grid, sim):
    progeny = Grid(grid.height, grid.width)
    item = next(sim)
    while item is not TICK:
        if isinstance(item, Query):
            state = grid.query(item.y, item.x)
            item = sim.send(state)
        else:  # Must be a Transition
            progeny.assign(item.y, item.x, item.state)
            item = next(sim)
    return progeny
```

```
for i in range(5):
    columns.append(str(grid))
    grid = live_a_generation(grid, sim)
```

Ch41. 진정한 병렬성을 실현하려면 concurrent.futures를 고려하자

파이썬 표준 구현체는 GIL로 인해 스레드가 병렬적으로 실행되지 않습니다.

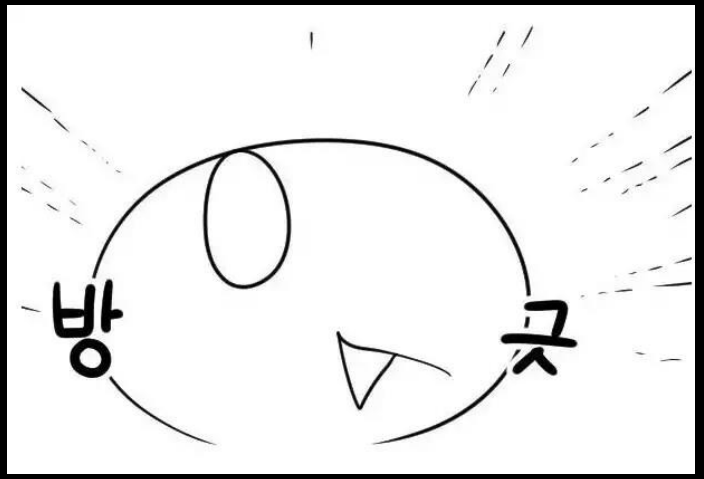
성능이 중요한 상황에서는 일반적으로 C언어로 모듈을 작성할 수 있지만 C로 코드를 재 작성하는 것은 상당한 비용이 듭니다.

저번 ppt가 떠오르는 군요…

파이썬에서 병렬성을 통해서 성능을 높이는 방법으로는 concurrent.futures를 사용하는 방법이 있습니다.

concurrent.futures는 자식 프로세스에서 추가적인 인터프리터를 실행해서 병렬로 CPU코어를 활용할 수 있게 합니다.

멀티 스레드가 안되면 멀티 프로세스



최대 공약수를 찾는 예제를 그냥 작성해 보면…

```
from time import time

def gcd(pair):
    a, b = pair
    low = min(a, b)
    for i in range(low, 0, -1):
        if a % i == 0 and b % i == 0:
            return i

numbers = [(1963309, 2265973), (2030677, 3814172), (1551645, 2229620), (2039045,
2020802)]
start = time()
result = list(map(gcd, numbers))
end = time()
print('Took %.3f seconds' % (end - start))

>>>
Took 1.507 seconds
```

concurrent.futures 의 ThreadPoolExecutor 를 사용해 보면…

```
start = time()
pool = ThreadPoolExecutor(max_workers=2)
result = list(pool.map(gcd, numbers))
end = time()
print('Took %.3f seconds' % (end - start))

>>>
Took 1.749 seconds
```

이건 다 프로세스 생성 비용 때문…

```
start = time()
pool = ProcessPoolExecutor(max_workers=2)
result = list(pool.map(gcd, numbers))
end = time()
print('Took %.3f seconds' % (end - start))

>>>
Took 0.939 seconds
```

생성한 프로세스를 Pool로 관리하여 재사용하도록 합니다.

매우 간단하지만 실제로는 복잡한 과정을 거칩니다.

1. 입력 데이터에서 map으로 각 아이템을 가져온다.
2. pickle 모듈을 사용하여 바이너리 데이터로 직렬화한다
3. 주 인터프리터 프로세스에서 직렬화한 데이터를 지역 소켓을 통해 자식 인터프리터 프로세스로 복사한다.
4. 자식 인터프리터에서는 pickle을 사용해서 데이터를 파이썬 객체로 역 직렬화한다.

5. gcd 함수가 들어 있는 파이썬 모듈을 임포트한다.
6. 다른 자식 프로세스를 사용하여 병렬로 입력 데이터를 처리한다.
7. 결과를 다시 바이트로 직렬화한다.
8. 소켓을 통해 바이트를 다시 복사한다.
9. 바이트를 부모 프로세스에 있는 파이썬 객체로 역 직렬화한다.
10. 여러 자식에 있는 결과를 하나의 결과로 합친다.

결국 **프로세스간 통신**이 이루어집니다.

전송해야 하는 데이터의 양은 적지만 많은 계산이 발생하는
지렛대 효과가 큰 작업에 적합합니다.

Reference

파이썬 코딩의 기술 – 저자 브렛 슬라킨 | 김형철옮김 | 길벗

전문가를 위한 파이썬 – 저자 루시아누 하말류 | 강권학옮김 | 한빛

라이프 게임 – https://goo.gl/AJn5F8

Thanks!