

2026.01.26.월

🕒 생성일	@2026년 1월 26일 오전 9:51
🏷 태그	Python

목차

1. 스레드 모듈
2. 스레드 풀
3. 프로세스
4. 비동기 프로그래밍

▼ 1) 스레드 모듈

- 스레드 모듈은 멀티스레딩 프로그래밍을 위한 고수준 인터페이스를 제공
- 주요 클래스
 - Thread

```
# 스레드 동기화 도구
import threading
import time

# 이벤트 객체 생성
event = threading.Event()

def waiter():
    print("대기자: 이벤트를 기다리는 중...")
    # 이벤트가 설정(set)될 때까지 이 지점에서 스레드가 대기합니다.
    event.wait()
    print("대기자: 이벤트를 수신하고 작업 진행!")

def setter():
    print("설정자: 작업 중...")
    # 특정 작업이 진행 중임을 가정하여 3초간 대기합니다.
    time.sleep(3)
    print("설정자: 이제 이벤트를 설정합니다.")
    # 이벤트를 설정하여 wait() 상태인 다른 스레드들을 깨웁니다.
    event.set()

# 스레드 생성 및 시작
t1 = threading.Thread(target=waiter)
t2 = threading.Thread(target=setter)

t1.start()
t2.start()
```

- 주요 매개변수
 - target : 스레드가 실행할 함수
 - args : target함수에 전달할 위치 인자 (튜플)
 - kwargs : target함수에 전달할 키워드 인자 (딕셔너리)
 - daemon : 데몬 스레드 여부 결정
 - True → 프로그램 종료시 데몬 스레드 강제 종료
 - False → 메인 프로그램이 데몬 스레드가 종료될 때까지 기다림

- name : 스레드의 이름 지정 (기본값: 자동 생성)
- 주요 메서드
 - start() : 스레드 시작
 - join([timeout]) : 스레드가 종료될 때까지 현재 스레드 대기, 시간을 지정하지 않으면 무한 대기
 - is_alive() : 스레드가 실행 중인지 확인 (불리언)
 - daemon(속성) : 스레드가 데몬인지 확인
 - name(속성) : 스레드의 이름을 가져오거나 설정
 - ident(속성) : 스레드의 고유 식별자 반환, 시작되지 않았으면 None
- 동기화 도구
 - set() : 이벤트 설정
 - clear() : 이벤트 해제
 - wait() : 이벤트가 설정될 때까지 대기
 - is_set() : 이벤트 상태 확인
- Lock : 동기화 도구
 - acquire() : 락 획득
 - release() : 락 해제
 - with 문 : 락 관리 가능
- RLock
 - 일반 락과 달리 같은 스레드가 여러 번 락을 획득할 수 있는 재진입 가능 락 (Reentrant Lock)
 - 획득한 만큼 해제해야 락이 완전히 해제됨
- Semaphore
 - 내부 카운터를 가진 동기화 도구
 - 지정한 값만큼의 스레드가 동시에 접근
 - 카운터가 0이면 대기
 - acquire() : 카운터 감소
 - release() : 카운터 증가
- Event
 - 스레드간 간단한 신호 메커니즘 제공
 - 내부 플래그 동작
 - set() : 플래그 True
 - clear() : 플래그 False
 - wait() : 플래그가 True가 될 때까지 대기
- Condition

```
import threading
import time

# 데이터와 Condition 객체 생성
data = None
condition = threading.Condition()

# 데이터를 기다리는 스레드
def wait_for_data():
    print("대기 스레드: 데이터를 기다립니다...")

    with condition: # Lock 획득
        condition.wait() # 데이터가 준비될 때까지 기다림
```

```

# 알림을 받으면 다시 Lock을 획득하고 계속 실행
print(f"대기 스레드: 데이터 '{data}'를 받았습니다!")
# 데이터를 준비하는 스레드

def prepare_data():
    global data

    print("준비 스레드: 데이터 준비 중...")
    time.sleep(2) # 데이터 준비 시간 시뮬레이션

    with condition: # Lock 획득
        data = "준비된 데이터"
        print("준비 스레드: 데이터가 준비되었습니다!")
        # 대기 중인 스레드 하나를 깨웁니다.
        condition.notify()

# 스레드 생성 및 시작
# t1은 데이터를 기다리고, t2는 데이터를 준비합니다.
t1 = threading.Thread(target=wait_for_data)
t2 = threading.Thread(target=prepare_data)

t1.start()
t2.start()

# 메인 스레드는 생성된 스레드들이 종료될 때까지 기다립니다.
t1.join()
t2.join()

```

- 특정 조건이 충족될 때까지 스레드를 대기 시킬 수 있다.
- 생산자-소비자 패턴에 자주 사용
- `acquire()` : 내부 Lock 획득
- `release()` : 내부 Lock 해제
- `wait()` : Lock 해제하고 알림
- `notify()` : 대기 중인 스레드 하나에 알림
- `notify_all()` : 모든 대기 스레드에 알림
- `join()` : 호출한 스레드가 완료될 때까지 메인 스레드 대기 (데몬, 일반 상관 없음)
- Barrider
 - 지정된 수의 스레드가 모두 특정 지점에 도달할 때까지 대기시키는 동기화 도구
- 작업 배분 시스템 ex

```

import threading
import queue
import time
import random

# 작업 큐 생성
task_queue = queue.Queue()
# 결과 큐 생성
result_queue = queue.Queue()

# 작업 생성 함수 (Producer 역할)
def create_tasks():
    print("작업 생성 시작")

    # 10개의 작업 생성

```

```

for i in range(10):
    task = f"작업-{i}"
    task_queue.put(task) # 작업 보관함(큐)에 넣음
    print(f"작업 추가: {task}")
    # 0.1~0.3초 사이의 랜덤한 간격을 두고 작업 생성
    time.sleep(random.uniform(0.1, 0.3))

# 영업 종료 신호 (워커 수만큼 추가)
# 'None'을 넣어 워커 스레드에게 더 이상 작업이 없음을 알립니다.
for _ in range(3): # 워커(요리사)가 3명임을 가정
    task_queue.put(None) # None은 "오늘 장사 끝" 신호

print("모든 작업 생성 완료")

# 작업 처리 함수
def worker(worker_id):
    print(f"워커 {worker_id} 시작") # 출근 신고
    while True: # 계속 일하는 무한 루프
        # 작업 가져오기 (주문서 확인)
        task = task_queue.get() # 보관함에서 작업 꺼냄

        # 퇴근 시간인지 확인
        if task is None: # "오늘 장사 끝" 신호 확인
            print(f"워커 {worker_id} 종료") # 퇴근!
            break # 무한 루프 종료

        # 작업 처리 (요리 만들기)
        print(f"워커 {worker_id}가 {task} 처리 중...")
        processing_time = random.uniform(0.5, 1.5) # 요리 시간은 랜덤
        time.sleep(processing_time) # 요리하는 시간

        # 결과 제출 (완성된 요리 올려두기)
        result = f"{task} 완료 (소요시간: {processing_time:.2f}초)"
        result_queue.put((worker_id, result)) # 결과 보관함에 넣음

        # 작업 완료 표시 (주문서에 완료 도장)
        # 현재 처리 중인 특정 작업 하나가 완료되었음을 큐에 알립니다.
        task_queue.task_done()
        print(f"남은 작업 수: {task_queue.qsize()}")

# 결과 수집 함수
def result_collector():
    print("결과 수집기 시작") # 서버 직원 출근
    results = [] # 완료된 주문 기록용

    # 총 10개 결과 수집 (10개 요리 서버)
    for _ in range(10):
        # 결과 큐에서 워커 ID와 결과 메시지를 꺼냅니다.
        worker_id, result = result_queue.get() # 완성된 요리 가져오기
        print(f"결과 수신: 워커 {worker_id} -> {result}")

        # 리스트에 결과를 추가하여 기록을 남깁니다.
        results.append(result) # 결과 기록

        # 해당 결과 처리가 완전히 끝났음을 큐에 알립니다.
        result_queue.task_done() # "이 요리 서버했어요" 표시

    print(f"총 {len(results)}개 결과 수집 완료") # 서버 완료

# 스레드 생성 및 시작

```

```

# 1. 작업을 만드는 생산자 스레드 생성
creator = threading.Thread(target=create_tasks)

# 2. 작업을 처리하는 워커 스레드들 생성 (3명의 워커)
workers = [threading.Thread(target=worker, args=(i,)) for i in range(3)]

# 3. 결과를 수집하는 컬렉터 스레드 생성
collector = threading.Thread(target=result_collector)

# 스레드 시작
creator.start()
for w in workers:
    w.start()
collector.start()

# 스레드 종료 대기
# 메인 스레드가 각 스레드의 작업이 끝날 때까지 기다립니다.
creator.join()
for w in workers:
    w.join()
collector.join()

print("모든 작업 완료!") # 오늘 영업 끝!

```

▼ 2) 스레드 풀

- 파이썬에서 스레드풀은 여러 개의 스레드를 미리 만들어두고 실행할 작업들을 이 풀에 던져서 효율적으로 처리
- `ThreadPoolExecutor` 를 사용하여 구현
 - `ThreadPoolExecutor(max_workers=None, thread_name_prefix='')`:
 - `max_workers` : 스레드 풀의 최대 스레드 수 (기본값은 CPU 코어 수 × 5)
 - `thread_name_prefix` : 생성된 스레드 이름의 접두사
- 새로운 스레드를 생성하는 것이 아닌 재사용하여 효율성 향상
- 오버헤드 감소 (만들어둔 스레드 재사용)
- 자원 관리 (스레드 수 상한선 제한)
- 생명 주기 관리 없이 간결한 코드

```

from concurrent.futures import ThreadPoolExecutor
import time

# 실행할 작업 함수
def fetch_data(name):
    print(f"[{name}] 데이터 다운로드 시작...")
    time.sleep(2) # I/O 작업을 시뮬레이션
    return f"[{name}] 완료 데이터"

# 1. 스레드풀 생성 (최대 스레드 3개)
with ThreadPoolExecutor(max_workers=3) as executor:
    # 2. 작업 제출 (map 사용 시 여러 인자를 한 번에 전달 가능)
    tasks = ["A", "B", "C", "D", "E"]
    results = executor.map(fetch_data, tasks)

# 3. 결과 출력
for result in results:
    print(result)

```

- 작업 제출 및 관리

- `submit(fn, *args, **kwargs)`
 - 작업 함수와 인자를 받아 작업 큐에 추가
 - Future 객체 반환 (비동기 작업의 상태와 결과를 추적)
- `map(fn, *iterables, timeout=None)`
 - 여러 인자에 동일 함수를 적용 (리스트 컴프리헨션과 유사)
 - 결과는 제출 순서대로 반환 (완료 순서가 아님)
- `shutdown(wait=True)`
 - 스레드 풀 종료
 - `wait=True` 면 모든 작업 완료까지 대기
- Future 객체
 - 작업의 상태를 실시간으로 확인 가능 (진행 중, 완료됨, 취소됨)
 - `done()` : 작업 완료 여부 확인 (True/False)
 - `result(timeout=None)` : 작업 결과 반환 (미완료 시 대기)
 - `cancel()` : 실행 전 작업 취소 시도
 - `add_done_callback(fn)` : 작업 완료 시 호출할 함수 등록
- 유틸리티 함수
 - `as_completed(futures, timeout=None)`
 - 완료되는 순서대로 Future 객체 반환 (제너레이터)
 - `wait(futures, timeout=None, return_when=ALL_COMPLETED)` *:
 - Future 객체들이 완료될 때까지 대기

```
import concurrent.futures
import random
import time

def fetch_data(url):
    print(f"{url} 데이터 요청 중...")
    # 데이터 요청 시간 시뮬레이션 (0.5~2초 사이 랜덤)
    time.sleep(random.uniform(0.5, 2))

    # 가끔 오류 발생 시뮬레이션 (20% 확률로 예외 발생)
    if random.random() < 0.2:
        raise Exception(f"{url} 연결 오류")

    # 성공 시 데이터 반환 (랜덤한 크기 정보 포함)
    return f"{url}의 데이터 (크기: {random.randint(100, 1000)}KB)"

# 요청할 URL 목록
urls = [
    "https://api.example.com/users",
    "https://api.example.com/products",
    "https://api.example.com/orders",
    "https://api.example.com/settings"
]

# 스레드 풀을 사용한 병렬 데이터 요청
with concurrent.futures.ThreadPoolExecutor(max_workers=3) as executor:
    # 각 URL에 대한 Future 객체 저장 (작업 제출)
    # 딕셔너리 컴프리헨션을 사용하여 Future 객체를 키로, URL을 값으로 저장합니다.
    future_to_url = {executor.submit(fetch_data, url): url for url in urls}

    # 완료된 순서대로 결과 처리
```

```

for future in concurrent.futures.as_completed(future_to_url):
    url = future_to_url[future] # 해당 Future 객체에 대응하는 URL 확인
    try:
        # 작업 결과를 가져옵니다. 에러가 발생했다면 여기서 raise 됩니다.
        data = future.result()
        print(f"성공: {url} -> {data}")
    except Exception as e:
        # 20% 확률로 발생하도록 설정했던 연결 오류 등을 여기서 처리합니다.
        print(f"실패: {url} -> {e}")

```

▼ 3) 프로세스

```

import multiprocessing
import time

def count_up(name, max_count):
    """숫자를 세는 간단한 함수"""
    for i in range(1, max_count + 1):
        print(f"프로세스 {name}: 카운트 {i}")
        time.sleep(0.5)

if __name__ == "__main__": # 중요: 항상 이 조건 필요
    # 프로세스 생성
    # 각각의 프로세스는 독립된 메모리 공간을 가집니다.
    p1 = multiprocessing.Process(target=count_up, args=("A", 5))
    p2 = multiprocessing.Process(target=count_up, args=("B", 3))

    # 프로세스 시작
    p1.start()
    p2.start()

    # 메인 프로세스에서 다른 프로세스 종료 대기
    p1.join()
    p2.join()

    print("모든 프로세스 종료!")

```

- 프로세스 생성
 - Process(group=None, target=None, name=None, args=(), kwargs={}, daemon=None)
 - Group : 항상 None (향후 확장을 위해 예약됨)
 - Target : 프로세스가 실행할 함수
 - Name : 프로세스 이름 (지정하지 않으면 'Process-N' 형식으로 자동 생성)
 - Args : target 함수에 전달할 위치 인자 (튜플)
 - Kwargs : target 함수에 전달할 키워드 인자 (딕셔너리)
 - Daemon : 데몬 프로세스 여부 (True/False)
- 주요 메서드
 - start() : 프로세스 실행 시작 (run() 메서드 호출)
 - run() : 프로세스 실행 시 호출되는 메서드 (기본적으로 target 함수 실행)
 - join([timeout]) : 프로세스가 종료될 때까지 대기
 - timeout : 최대 대기 시간 (초), 지정하지 않으면 무한 대기
 - is_alive() : 프로세스가 현재 실행 중인지 확인 (True/False 반환)
 - terminate() : 프로세스 강제 종료 (정상적인 종료가 아님)
 - kill() : 프로세스 즉시 강제 종료 (terminate보다 더 강력)

- close() : 프로세스와 관련된 리소스 정리

- Lock 활용

```
import multiprocessing
import time

def add_to_shared(shared_value, lock, increment):
    """공유 자원에 값을 더하는 함수"""
    for _ in range(5):
        # 락(Lock)을 사용하여 여러 프로세스가 동시에 수정하지 못하도록 동기화합니다.
        with lock: # 락 사용하여 동기화
            shared_value.value += increment

        # 각 작업 사이에 짧은 지연 시간을 줍니다.
        time.sleep(0.1)

    # 현재 작업을 수행 중인 프로세스 이름을 출력하며 종료를 알립니다.
    print(f"프로세스 {multiprocessing.current_process().name}: 작업 완료")

if __name__ == "__main__": # 윈도우/macOS에서 프로세스 실행 시 필수
    # 공유 변수 생성 (초기값 0의 정수), 여러 프로세스가 접근할 수 있게 함
    shared_number = multiprocessing.Value('i', 0)

    # 공유 변수 접근을 위한 락 생성
    # 여러 프로세스가 동시에 수정하여 데이터가 꼬이는 것을 방지합니다.
    lock = multiprocessing.Lock()

    # 프로세스 생성
    # p1은 1씩 5번, p2는 2씩 5번 더하는 작업을 각각 수행하도록 설정합니다.
    p1 = multiprocessing.Process(target=add_to_shared, args=(shared_number, lock, 1))
    p2 = multiprocessing.Process(target=add_to_shared, args=(shared_number, lock, 2))

    # 프로세스 시작
    p1.start()
    p2.start()

    # 프로세스 종료 대기
    # 각 프로세스의 작업이 끝날 때까지 메인 프로세스가 기다립니다.
    p1.join()
    p2.join()

    # 최종 결과 출력
    # 공유 자원의 실제 값에 접근할 때는 .value 속성을 사용합니다.
    print(f"최종 공유 값: {shared_number.value}") # 예상 결과: 15 (5*1 + 5*2)
```

- Queue 활용

```
import multiprocessing
import time
import random

# Queue를 이용한 통신 예제
def producer_process(queue):
    print(f"생산자 프로세스 시작: {multiprocessing.current_process().name}")
    for i in range(5):
        item = f"데이터-{i}"
        queue.put(item) # 큐에 항목 추가
        print(f"생산: {item}")
        time.sleep(random.uniform(0.1, 0.5))
```

```

# 작업 완료 신호 (Poison Pill)
queue.put(None)
print("생산자 프로세스 종료")

def consumer_process(queue):
    print(f"소비자 프로세스 시작: {multiprocessing.current_process().name}")
    while True:
        # 큐에서 데이터를 하나씩 꺼냅니다.
        item = queue.get()

        if item is None: # 종료 신호 확인
            break

        print(f"소비: {item}")
        time.sleep(random.uniform(0.2, 0.7))

    print("소비자 프로세스 종료")

if __name__ == "__main__": # 윈도우에서 프로세스 실행 시 필요
    # 프로세스 간 통신을 위한 큐 생성
    q = multiprocessing.Queue()

    # 프로세스 생성
    # 큐 객체(q)를 각 프로세스의 인자로 전달하여 소통 창구를 마련합니다.
    prod = multiprocessing.Process(target=producer_process, args=(q,))
    cons = multiprocessing.Process(target=consumer_process, args=(q,))

    # 프로세스 시작
    prod.start()
    cons.start()

    # 프로세스 종료 대기
    # 생산자와 소비자의 작업이 모두 끝날 때까지 메인 프로세스가 대기합니다.
    prod.join()
    cons.join()

    print("모든 프로세스 종료")

```

- Pool 활용

```

import multiprocessing
import time
import os

# 병렬 처리할 작업 함수
def process_task(task_id):
    # 현재 작업을 수행 중인 프로세스의 고유 ID를 가져옵니다.
    process_id = os.getpid()
    print(f"작업 {task_id} 시작 (프로세스 ID: {process_id})")

    # CPU 작업 시뮬레이션: 무거운 연산 작업을 수행합니다.
    result = 0
    for i in range(10000000): # 1,000만 번의 반복 연산
        result += i

    print(f"작업 {task_id} 완료 (프로세스 ID: {process_id})")

# 작업 번호, 계산 결과, 프로세스 ID를 반환합니다.

```

```

return task_id, result, process_id

if __name__ == "__main__": # 윈도우/macOS에서 프로세스 실행 시 필수
    # 시스템의 CPU 코어 수 확인
    num_cores = multiprocessing.cpu_count()
    print(f"시스템 CPU 코어 수: {num_cores}")

    # 작업 목록 생성 (0부터 9까지 총 10개의 작업)
    tasks = range(10)

    # 1. 순차 처리 시간 측정
    start_time = time.time()
    # 리스트 컴프리헨션을 사용하여 하나씩 차례대로 실행합니다.
    sequential_results = [process_task(i) for i in tasks]
    end_time = time.time()
    print(f"순차 처리 시간: {end_time - start_time:.2f}초")

    # 2. 병렬 처리 시간 측정
    start_time = time.time()
    # Pool 객체를 생성하여 작업을 모든 CPU 코어에 골고루 배분합니다.
    with multiprocessing.Pool(processes=num_cores) as pool:
        parallel_results = pool.map(process_task, tasks)
    end_time = time.time()
    print(f"병렬 처리 시간: {end_time - start_time:.2f}초")

    # 사용된 프로세스 ID 확인
    # 각 작업 결과에서 저장된 PID를 추출하여 중복을 제거합니다.
    process_ids = set(result[2] for result in parallel_results)
    print(f"사용된 프로세스 수: {len(process_ids)}")
    print(f"프로세스 ID 목록: {process_ids}")

```

▼ 4) 비동기 프로그래밍

- 이벤트 루프(Event Loop): 비동기 프로그래밍의 사령탑입니다. 여러 작업들을 관찰하다가, 어떤 작업이 준비(데이터 도착 등)되면 해당 작업을 실행시키는 역할을 합니다.
- 코루틴(Coroutine): 일반적인 함수와 달리, 중간에 실행을 멈췄다가(await) 나중에 다시 재개할 수 있는 특수한 함수입니다. 파이썬에서는 async def로 정의합니다.
- 태스크(Task): 루프에서 실행할 코루틴을 스케줄링하는 단위입니다.
- 네트워크 요청(API 호출), 데이터베이스 쿼리, 파일 읽기/쓰기는 CPU 연산보다 더 많은 시간이 소모되고 대기 시간 동안 다른 연산을 수행하여 효율성을 높인다.

```

import asyncio

# 코루틴 정의
async def say_hello(name, delay):
    print(f"{name} 인사 시작")
    # 비동기적으로 대기하며, 이 동안 다른 작업이 실행될 수 있도록 제어권을 넘깁니다.
    await asyncio.sleep(delay) # 비동기적으로 대기 (블로킹하지 않음)
    print(f"{name} 인사 완료 (대기 시간: {delay}초)")
    return f"{name}의 결과"

# 메인 코루틴
async def main():
    print("프로그램 시작")

    # 여러 코루틴을 동시에 실행하고 결과를 모읍니다.
    # gather를 사용하면 가장 긴 대기 시간(3초) 내에 모든 작업이 완료됩니다.
    results = await asyncio.gather(

```

```
    say_hello("A", 3),
    say_hello("B", 1),
    say_hello("C", 2)
)
```

```
print(f"모든 결과: {results}")
print("프로그램 종료")
```

```
# 이벤트 루프 생성 및 실행
if __name__ == "__main__":
    asyncio.run(main())
```

- `create_task()` : 코루틴을 태스크로 변환하여 병렬 실행
- `gather()` : 여러 코루틴/태스크를 동시에 실행하고 모든 결과 반환
- `wait()` : 여러 태스크가 완료될 때까지 대기
- `wait_for()` : 시간 제한과 함께 코루틴 완료 대기