Примитивы синхронизации

Теперь мы с вами познакомимся с механизмами, которые ограничивают доступ к ресурсам и возможностям внутри асинхронного приложения.

Обратите внимание, что все эти способы синхронизации не предназначены для использования внутри тредов.

Lock

Простейший механизм синхронизации, аналог **Mutex** в других языках программирования.

Пример. У нас есть пачка чипсов. Единовременно только 1 рука может находиться внутри.

Lock гарантирует эксклюзивный доступ к общим ресурсам, в нашем случае - к чипсам)

```
import asyncio



class PackOfChips:
    def __init__(self):
        self._lock = asyncio.Lock()

    async def reserve(self):
        await self._lock.acquire()

    def cheeps(self):
        self._lock.release()



async def handle(name: str, pack: PackOfChips):
    await pack.reserve() # первая короутина, которая дошла до этого места заблокирует доступ и продолжит выполнение, а остальные - ждать освобождения ресурсов
    print(f"{name} put hand in pack")
    await asyncio.sleep(1) # сон показывает, что все остальные короутины заблокированы на await pack.reserve()
    pack.cheeps() # освобождаем ресурсы
    print(f"{name} took cheeps")



async def main():
    people = ["Andrey", "Alex", "Artem", "Igor"]
    pack = PackOfChips()

    await asyncio.gather(*[handle(name, pack) for name in people]) # запускаем параллельно несколько короутин

asyncio.run(main())
```

```
Andrey put hand in pack
Andrey took cheeps
Alex put hand in pack
Alex took cheeps
Artem put hand in pack
Artem took cheeps
Igor put hand in pack
Igor took cheeps
```

Event

Event используется для сигнализации корутин о неком событии.

Пусть у нас есть 2 функции:

run - она ждет события с помощью "await event.wait()"

wait_and_set - порождает событие и сигнализирует об этом ожидающую корутину.

```
import asyncio

async def wait_and_set(event: asyncio.Event):
    await asyncio.sleep(1)
    event.set()

async def run():
    event = asyncio.Event()
    asyncio.create_task(wait_and_set(event))
    await event.wait()

if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    loop.run_until_complete(run())
```

В итоге при запуске кода получим:

1. запускается корутина **wait_and_set** и засыпает на секунду
2. основная корутина **run** начинает ждать на событии event
3. корутина **wait_and_set** просыпается через 1 секунду и вызывает **event.set()**
4. основная корутина **run** просыпается и программа завершается

Если вызвать **event.set()**, а затем сразу **await event.wait()**, то событие уже произойдет и выполнение не остановится.

Рассмотрим чуть более сложный пример: допустим нам нужно написать планировщик, который просыпается раз в период времени **timer** и запускает **_worker**.

На планировщик налагаются дополнительные условия:

- **_worker** может выполняться дольше чем период времени time, при этом планировщик не должен сбиваться
- при остановке планировщика нужно дождаться выполнения всех запущенных задач

```
import asyncio
import datetime
from typing import Optional



class Scheduler:
    def __init__(self, timer: int):
        self.timer = timer
        self.is_running = True
        self._stop_event = asyncio.Event()
        self._scheduler_task: Optional[asyncio.Task] = None
        self._concurrent_workers = 0

    async def _worker(self):
        self._concurrent_workers += 1

        print('start', datetime.datetime.now())
        await asyncio.sleep(5)  # какая-то полезная работа, которая занимает 5 секунд
        print('stop', datetime.datetime.now())

        self._concurrent_workers -= 1
        if not self.is_running and self._concurrent_workers == 0:
            # если планировщик остановлен и при этом это был последний запущенный worker, то нужно уведомить
            # корутину stop о том, что все _worker завершились
            self._stop_event.set()

    async def _scheduler(self):
        """
        планировщик запускается в фоне и просыпается раз в период времени timer
        """
        while self.is_running:
            asyncio.create_task(self._worker())
            await asyncio.sleep(self.timer)

    def start(self):
        self.is_running = True
        self._scheduler_task = asyncio.get_event_loop().create_task(self._scheduler())

    async def stop(self):
        """
        ставим is_running = False, чтобы планировщик не планировал новые запуски и отменяем его задачу
        ждем пока все _worker завершаться
        """
        self.is_running = False
        self._scheduler_task.cancel()
        await self._stop_event.wait()



if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    s = Scheduler(2)
    s.start()
    try:
        loop.run_forever()
    except KeyboardInterrupt:
        loop.run_until_complete(s.stop())
```

Semaphore

Еще один способ синхронизации, но завязанный на счетчик - **Semaphore**.

Пример. Хотим ограничить количество единовременных обработок запросов в базу некоторым числом — например, 10.

Код этой блокировки будет выглядеть следующим образом:

```
import asyncio
import time



class Query:
    async def execute(self):
        await asyncio.sleep(1)



class PGStore:
    def __init__(self):
        self._sem = asyncio.Semaphore(2) # ограничиваем число одновременных доступов к ресурсам до 2
        self._init_time = time.time()

    async def execute_request(self, query):
        async with self._sem: # каждое использование контекстного менеджера будет уменьшать счетчик семафора и когда счетчик станет равным 0 - короутины, ранее не выполнившие async with self._sem - заблокируются
            print(f"{time.time() - self._init_time} request was started")
            await query.execute()
            print(f"{time.time() - self._init_time} request was handled")



async def main():
    store = PGStore()
    coros = [store.execute_request(Query()) for _ in range(100)]

    await asyncio.gather(*coros)



asyncio.run(main())
```

```
0.0004100799560546875 request was started
0.0009899139404296875 request was started
1.0068840980529785 request was handled
1.007072925567627 request was handled
1.007131814956665 request was started
1.0072309970855713 request was started
2.0126938819885254 request was handled
2.0128917694091797 request was handled
```

По выводу можем заметить, что в 1 и ту же секунду обрабатывались только 2 запроса