ЛЕКЦІЯ 7
БАГАТОПОТОЧНІСТЬ В Python

Способи реалізації паралельних обчислень у програмах на Python.

Що таке паралелізм?

Паралелізм дозволяє працювати над кількома обчисленнями одночасно в одній програмі. Такої поведінки в Python можна досягти кількома способами:

- Використовуючи багатопоточність threading, дозволяючи кільком потокам працювати по черзі.
- Використовуючи кілька ядер процесорів, multiprocessing.. Робити відразу кілька обчислень, використовуючи кілька ядер процесора. Це називається паралелізмом.
- Використовуючи асинхронне введення-виведення з модулем asyncio. Запускаючи якесь завдання, продовжувати робити інші обчислення замість очікування відповіді від мережевого підключення або від операцій читання/запису.

Різниця міжд потоками та процесами.

Потік threading - це незалежна послідовність виконання якихось обчислень. Потік thread ділить виділену пам'ять ядру процесора, а також його процесорний час з усіма іншими потоками, які створюються програмою в рамках одного ядра процесора. Програми на мові Python мають один основний потік. Можна створити їх більше і дозволити Python перемикатися між ними. Це перемикання відбувається дуже швидко і здається, що вони працюють паралельно.

Поняття **процесу в multiprocessing -** являє собою незалежну послідовність виконання обчислень. На відміну від потоків threading, процес має власне ядро і, отже, виділену йому пам'ять, яке не використовується спільно з іншими процесами. Процес може клонувати себе, створюючи два або більше екземплярів в одному ядрі процесора.

Асинхронне введення-виведення не є ні потоковим (threading), ні багатопроцесорним (multiprocessing). Власне, це однопоточна, однопроцессная парадигма і належить до паралельним обчислень.

Python має одну особливість, яка ускладнює паралельне виконання коду. Вона називається GIL, скорочено від Global Interpreter Lock. GIL гарантує,

що у будь-який момент часу працює лише один потік. З цього випливає, що з потоками неможливо використовувати кілька ядер процесора.

GIL був введений у Python тому, що керування пам'яттю CPython не є потокобезпечним. Маючи таке блокування Python може бути впевненим, що ніколи не буде умов гонки.

Що таке умови гонки та потокобезпека?

• Стан гонки виникає, коли кілька потоків можуть одночасно отримувати доступ до загальної структури даних або місцезнаходження в пам'яті та змінювати їх, внаслідок чого можуть статися непередбачувані речі.

Приклад із життя: якщо два користувача одночасно редагують той самий документ онлайн і другий користувач збереже дані в базу, то перезапише роботу першого користувача. Щоб уникнути умов гонки, необхідно змусити другого користувача чекати, поки перший закінчить роботу з документом і лише після цього дозволити другому користувачеві відкрити та почати редагувати документ.

• Потокобезпека працює шляхом створення копії локального сховища у кожному потоці, щоб дані не стикалися з іншим потоком.Алгоритм планирования доступа потоков к общим данным.

Як мовилося раніше, потоки використовують одну й ту ж виділену пам'ять. Коли кілька потоків працюють одночасно, то не можна вгадати порядок, у якому потоки будуть звертатися до загальних даних. Результат доступу до даних, що спільно використовуються, залежить від алгоритму планування. який вирішує, який потік та коли запускати. Якщо такого алгоритму немає, то кінцеві дані можуть бути не такими, як очікуєш.

Наприклад, є загальна змінна a = 2. Тепер припустимо, що є два потоки, thread_one та thread_two. Вони виконують такі операції:a = 2

```
# функція 1 потоку
def thread_one():
    global a
    a = a + 2

# функція 2 потоку
def thread_two():
    global a
```

```
    a = a * 3
```

Якщо потік thread_one отримає доступ до спільної змінної a першим і thread_two другим, то результат буде 12:

1. 2 + 2 = 4;
2. 4 * 3 = 12.

Або навпаки, спочатку запуститься thread_two, а потім thread_one, то ми отримаємо інший результат:

1. 2 * 3 = 6;
2. 6 + 2 = 8.

Таким чином, очевидно, що порядок виконання операцій потоками має значення

Без алгоритмів планування доступу потоків до загальних даних такі помилки дуже важко знайти і налагодити. Крім того, вони, як правило, відбуваються випадковим чином, викликаючи безладну та непередбачувану поведінку.

Є ще найгірший варіант розвитку подій, який може статися без вбудованого в Python блокування потоків GIL. Наприклад, якщо обидва потоки починають читати глобальну змінну a одночасно, обидва потоки побачать, що a = 2, а далі, залежно від того який потік здійснить обчислення останнім, зрештою і дорівнюватиме змінна a (4 або 6). Не те, що очікувалося!

Дослідження різних підходів до паралельних обчислень у Python.

Визначимо функцію, яку використовуватимемо для порівняння різних варіантів обчислень. У всіх наступних прикладах використовується та сама функція, звана heavy() :

```
def heavy(n):
  for x in range(1, n):
    for y in range(1, n):
      x**y
```

Функція heavy() є вкладеним циклом, який виконує піднесення до степіня. Ця функція пов'язана зі швидкістю ядра процесора для

здійснення математичних обчислень. Якщо спостерігати за операційною системою під час виконання функції, можна побачити завантаження ЦП близьке до 100%.

Будемо запускати цю функцію по-різному, тим самим досліджуючи різницю між звичайною однопоточною програмою Python, багатопоточністю та багатопроцесорністю.

Однопоточний режим работи.

Кожна програма Python має, принаймні, один основний потік. Нижче наведено приклад коду для запуску функції heavy() в одному основному потоці одного ядра процесора, який виконує всі операції послідовно і буде еталоном з точки зору швидкості виконання:

```
import time

def heavy(n):
    for x in range(1, n):
        for y in range(1, n):
            x**y

def sequential(n):
    for i in range(n):
        heavy(500, i)
    print(f"{n} циклов вычислений закончены")

if __name__ == "__main__":
    start = time.time()
    sequential(80)
    end = time.time()
    print("Общее время работы: ", end - start)

# 80 циклов вычислений закончены
# Общее время работы:  23.573118925094604
```

Використання потоків `threading`.

У наступному прикладі будемо використовувати кілька потоків для виконання функції heavy(). Також зробимо 80 циклів обчислень. Для

цього розділимо обчислення на 4 потоки, у кожному з яких запустимо 20 циклів:

```
import threading
import time

def heavy(n, i, thead):
    for x in range(1, n):
        for y in range(1, n):
            x**y
    print(f"Цикл № {i}. Поток {thead}")

def sequential(calc, thead):
    print(f"Запускаем поток № {thead}")
    for i in range(calc):
        heavy(500, i, thead)
    print(f"{calc} циклов вычислений закончены. Поток № {thead}")


def threaded(theads, calc):
    # theads - количество потоков
    # calc - количество операций на поток

    threads = []

    # делим вычисления на 4 потока
    for thead in range(theads):
        t = threading.Thread(target=sequential, args=(calc,
thead))
        threads.append(t)
        t.start()

    # Подождем, пока все потоки
    # завершат свою работу.
    for t in threads:
        t.join()

if __name__ == "__main__":
    start = time.time()
    # разделим вычисления на 4 потока
```

```
    # в каждом из которых по 20 циклов
    threaded(4, 20)
    end = time.time()
    print("Общее время работы: ", end - start)

# Весь вывод показывать не будем
# ...
# ...
# ...
# Общее время работы:  43.33752250671387
```

Однопотоковий режим роботи, виявився майже в 2 рази швидше, тому що один потік не має накладних витрат на створення потоків (у нашому випадку створюється 4 потоки) та перемикання між ними.

Якби у Python не було GIL, то обчислення функції heavy() відбувалися швидше, а загальний час виконання програми прагнув до часу виконання однопоточної програми. Причина, через яку багатопотоковий режим в даному прикладі не буде працювати швидше за однопотоковий - це обчислення, пов'язані з процесором і полягають у GIL!

Якби функція heavy() мала багато блокуючих операцій, таких як мережеві виклики або операції з файловою системою, то застосування багатопотокового режиму роботи було б виправдане і дало б величезне збільшення швидкості!

Це твердження можна перевірити змоделювавши операції введення-виведення за допомогою функції time.sleep().

```
import threading
import time

def heavy():
    # имитации операций ввода-вывода
    time.sleep(2)

def threaded(theads):
    threads = []

    # делим операции на 80 потоков
    for thead in range(theads):
        t = threading.Thread(target=heavy)
```

```
        threads.append(t)
        t.start()

    # Подождем, пока все потоки
    # завершат свою работу.
    for t in threads:
        t.join()
    print(f"{theads} циклов имитации операций ввода-вывода
закончены")

if __name__ == "__main__":
    start = time.time()
    # 80 потоков - это неправильно и показано
    # чисто в демонстрационных целях
    threaded(80)
    end = time.time()
    print("Общее время работы: ", end - start)


# 80 циклов имитации операций ввода-вывода закончены
# Общее время работы:  2.008725881576538
```

Навіть якщо уявний введення-виведення ділиться на 80 потоків і всі вони будуть спати протягом двох секунд, то код все одно завершиться трохи більше ніж за дві секунди, тому що багатопотоковій програмі потрібен час на планування та запуск потоків.

Примітка! Кожен процесор підтримує певну кількість потоків на ядро, закладене виробником, у яких працює оптимально швидко. Не можна створювати безліч потоків. При збільшенні кількості потоків на величину, більшу, ніж заклав виробник, програма буде виконуватися довше або взагалі поведеться непередбачуваним чином (аж до зависання).

Використання багатопроцесорної обробки `multiprocessing`.

Тепер спробуємо справжню паралельну обробку з допомогою модуля multiprocessing. Модуль multiprocessing багато в чому повторює API модуля threading, тому зміни в коді будуть незначні.

Для того, щоб зробити 80 циклів обчислень функції heavy(), дізнаємося скільки процесор має ядер, а потім поділимо цикли обчислень на кількість ядер.

```
import multiprocessing
import time

def heavy(n, i, proc):
    for x in range(1, n):
        for y in range(1, n):
            x**y
    print(f"Цикл № {i} ядро {proc}")

def sequential(calc, proc):
    print(f"Запускаем поток № {proc}")
    for i in range(calc):
        heavy(500, i, proc)
    print(f"{calc} циклов вычислений закончены. Процессор № 
{proc}")


def processesed(procs, calc):
    # procs - количество ядер
    # calc - количество операций на ядро

    processes = []

    # делим вычисления на количество ядер
    for proc in range(procs):
        p = multiprocessing.Process(target=sequential, 
args=(calc, proc))
        processes.append(p)
        p.start()

    # Ждем, пока все ядра
    # завершат свою работу.
    for p in processes:
        p.join()

if __name__ == "__main__":
```

```
    start = time.time()
    # узнаем количество ядер у процессора
    n_proc = multiprocessing.cpu_count()
    # вычисляем сколько циклов вычислений будет приходится
    # на 1 ядро, что бы в сумме получилось 80 или чуть больше
    calc = 80 // n_proc + 1
    processesed(n_proc, calc)
    end = time.time()
    print(f"Всего {n_proc} ядер в процессоре")
    print(f"На каждом ядре произведено {calc} циклов вычислений")
    print(f"Итого {n_proc*calc} циклов за: ", end - start)


# Весь вывод показывать не будем
# ...
# ...
# ...
# Всего 6 ядер в процессоре
# На каждом ядре произведено 14 циклов вычислений
# Итого 84 циклов вычислений за:  5.0251686573028564
```

Код виконався майже вп'ятеро швидше. Це чудово демонструє лінійне збільшення швидкості обчислень кількості ядер процесора.

Використання багатопроцесорної обробки із пулом.

Можна зробити попередню версію програми трохи елегантнішою, використовуючи multiprocessing.Pool(). Об'єкт пулу, управляє пулом робочих процесів, у який можуть бути відправлені завдання. Використовуючи метод Pool.starmap(), можна провести ініціалізацію функції sequential() кожного процесу.
З метою експерименту в функції запуску пулу процесів pooled(core)передбачено ручне вказування кількості ядер процесора. Якщо не вказувати значення core, то за замовченням використовуватиметься кількість ядер процесора вашої системи, що є розумним вибором:

```
import multiprocessing
import time
```

```
def heavy(n, i, proc):
    for x in range(1, n):
        for y in range(1, n):
            x**y
    print(f"Вычисление № {i} процессор {proc}")


def sequential(calc, proc):
    print(f"Запускаем поток № {proc}")
    for i in range(calc):
        heavy(500, i, proc)
    print(f"{calc} циклов вычислений закончены. Процессор №
{proc}")


def pooled(core=None):
    # вычисляем количество ядер процессора
    n_proc = multiprocessing.cpu_count() if core is None else
core
    # вычисляем количество операций на процесс
    calc =  int(80 / n_proc) if 80 % n_proc == 0 else int(80 //
n_proc + 1)
    # создаем список инициализации функции
    # sequential(calc, proc) для каждого процесса
    init = map(lambda x: (calc, x), range(n_proc))
    with multiprocessing.Pool() as pool:
       pool.starmap(sequential, init)

    print (calc, n_proc, core)
    return (calc, n_proc, core)

if __name__ == "__main__":
    start = time.time()
    # в целях эксперемента, укажем количество
    # ядер больше чем есть на самом деле
    calc, n_proc, n = pooled(20)
    end = time.time()
    text = '' if n is None else 'задано '
    print(f"Всего {text}{n_proc} ядер процессора")
    print(f"На каждом ядре произведено {calc} циклов вычислений")
```

```
    print(f"Итого {n_proc*calc} циклов за: ", end - start)


# Весь вывод показывать не будем
# ...
# ...
# ...
# Всего задано 20 ядер процессора
# На каждом ядре произведено 4 циклов вычислений
# Итого 80 циклов за:  5.422096252441406
```

З результатів роботи видно, що час роботи трохи збільшився.

Якщо запустити цей код, можна простежити, що обчислення однаково відбуваються на тій кількості ядер, яка у процесорі. Тільки обчислення відбуваються по черзі - через це незначне збільшення часу роботи програми.

Висновки:

• Використовуйте модулі threading або asyncio для програм, пов'язаних із введенням-виводом, щоб значно підвищити продуктивність.
• Використовуйте модуль multiprocessing для вирішення проблем, пов'язаних із операціями ЦП. Цей модуль використовує весь потенціал всіх ядер у процесорі.