Прикладне програмування

ТЕМА 4. ПАРАЛЕЛЬНІ ОБЧИСЛЕННЯ

Паралельні обчислення

Паралельні обчислення – це виконання двох або більше задач у відрізки часу, що перетинаються.

У англомовні термінології є 2 терміни: concurrency та parallelism.

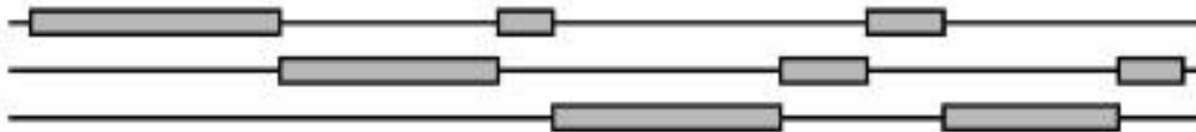
У російськомовній або україномовній термінології вони об'єднані терміном паралельні обчислення.

Parallelism означає, що 2 або більше задач виконуються одночасно, наприклад на багатоядерному процесорі.

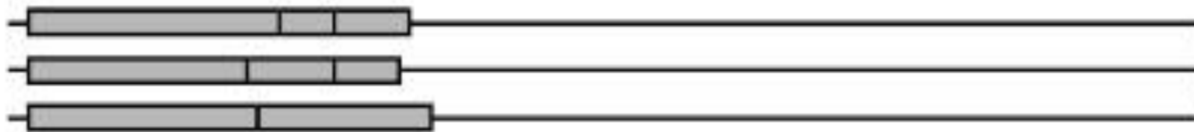
Є якісне, але неточне тлумачення concurrency та parallelism. concurrency – коли 2 черги обслуговує 1 касир, parallelism – коли 2 черги обслуговують 2 касири (кожен –свою).

Паралельні обчислення.2

Concepts in Concurrency



Concurrent, non-parallel execution



Concurrent, parallel execution

Процеси та потоки

Процес – це програма, що виконується.

Потік – це частина програми, що виконується, яка здатна виконуватись паралельно з іншими потоками.

Кожен процес під час виконання має свою власну пам’ять.

Усі потоки, що виконуються у рамках процесу, мають спільну пам’ять.

Багатопотоковість у Python. GIL

Слід зауважити, що у Python одночасно може виконуватись тільки 1 версія інтерпретатора.

Тобто, виконання декількох потоків саме програмного коду Python не є можливим.

За цим слідкує так зване глобальне блокування інтерпретатора (GIL – Global Interpreter Lock).

Коли потік починає виконуватись, він «отримує» GIL від Python, і надалі жоден інший потік не може виконуватись до звільнення GIL.

Але одночасне виконання декількох потоків можливе, якщо виклики виводять за межі коду Python, наприклад, до коду, написаного у C/C++.

Багатопотоковість у Python. GIL.2

Кожен потік Python після його запуску зупинити ззовні неможливо (окрім ситуації завершення самого процесу, у рамках якого виконується потік).

Потоки мають завершуватись самостійно, або продовжувати виконуватись до кінця роботи процесу.

Якщо один з потоків не закінчився, то це може затримати закінчення всього процесу.

Для перемикання між потоками Python використовує метрики щодо виконаної кількості команд.

Якщо перевищено встановлений поріг, - керування передається одному з активних потоків.

Інший спосіб – це передача управління самим потоком, наприклад за допомогою функції time.sleep

Потоки. Засоби роботи з потоками у Python

Є декілька модулів, які забезпечують програмування потоків:

- threading
- concurrent.futures

Модуль threading містить декілька класів для організації багатопотокового виконання програми

Клас Thread

Клас Thread призначено для запуску потоку з певними параметрами, а також для повернення/зміни характеристик потоку.

Конструктор класу виглядає так:

```
Thread(group=None, target=None, name=None, args=(), kwargs={}, *, daemon=None)
```

target – функція, яку слід віконувати у потоці

args, kwargs – позиційні та ключові параметри цієї функції.

Клас Thread.2

Методи класу Thread зібрано у таблиці:

Метод	Опис
start()	Починає роботу у окремому потоці
run()	Викликається після виклику start. Призначений для перевизначення у нащадках
join()	Чекати на завершення потоку

Клас Thread.3

Для підготовки та запуску у потоці функції fun з 2 параметрами, можемо написати:

```
th = Thread(target=fun, args=(5, ), kwargs={'b': 7})
th.start()
```

Приклад: простий потік для обчислення факторіалу, версії 0, 1

Обчислимо факторіал у окремому потоці.

Покажемо процес роботи основного потоку та потоку обчислення факторіалу.

У версії 0 бачимо змішування повідомлень від потоків при виведенні

Потоки–«демони»

Потоки–«демони» призначені для виконання у фоновому режимі.

У Python такі потоки відрізняються від звичайних зокрема тим, що завершуються при завершенні процесу.

Для вказання потоку-демону потрібно при створенні об’єкту класу Thread вказати параметр

daemon=True

Наприклад,

```
th = Thread(target=fun, args=(5, ), kwargs={'b': 7}, daemon=True)
```

Приклад: простий потік для обчислення факторіалу, версія 2

Обчислимо факторіал у окремому потоці.

Покажемо процес роботи основного потоку та потоку обчислення факторіалу.

У версії 2 визначимо потік як потік-демон.

Наслідування від Thread

Для того, щоб утворити клас-нащадок Thread, як правило, перевизначають конструктор __init__ та метод run.

Звичайно, у класі-нащадку можуть додаватись власні методи.

Клас-нащадок Thread може бути корисним, наприклад, для повернення результату функції після її завершення у потоці.

Приклад: простий потік для обчислення факторіалу, версія 3

Обчислимо факторіал у окремому потоці.

Покажемо процес роботи основного потоку та потоку обчислення факторіалу.

У версії 3 опишемо клас-нащадок Thread, у якому будемо повертати результат обчислень.

Використання потоками спільних даних

Спільні дані потоків – це змінні або поля класів, які можуть бути доступні декільком потокам.

У розглянутому прикладі потоки не використовували спільних даних. Але це є скоріше винятком, ніж правилом. Потоки можуть мати і у багатьох випадках мають спільні дані.

Використання спільних даних можна розділити на читання даних та зміну даних.

Якщо потоки тільки читають спільні дані, це практично не відрізняється від поведінки однопотокових програм.

Різниця дається взнаки при зміні спільних даних.

Порядок доступу до різних потоків до спільних даних є недетермінованим, що може спричиняти проблеми.

Приклад: додавання одиниць, версія 1

Розглянемо програму, що містить 2 потоки, які використовують одну і ту ж функцію для обчислення певного числа шляхом додавання одиниць.

Результат додавання будемо зберігати у змінній, що доступна обом потокам.

Подивимось на виконання такого обчислення за декілька сеансів роботи потоків.

Ситуація змагальності(race condition)

Ситуація змагальності (ще називають гонитва за даними) – це ситуація, у якій декілька потоків одночасно та несинхронізовано змінюють спільні дані.

Проблема у виконанні попереднього прикладу полягає у тому, що будь-який з 2 потоків може читати значення спільної змінної (з метою її подальшої зміни) у той час, коли інший потік впроваджує таку зміну.

Ситуація змагальності однозначно є небажаною та її треба вирішувати.

Синхронізація потоків

Вирішення проблем з гонитвою за даними – це використання надійних засобів синхронізації потоків, що є стійкими для виконання у багатопотоковому середовищі.

Щоб виключити гонитву за даними ми маємо тимчасово обмежувати доступ потоків до спільних даних.

Таке обмеження часто роблять за допомогою блокування та «замків».

У загальній термінології використовують термін мьютекс (mutex – mutually exclusive), що означає виключення можливості виконання інших потоків у моменти використання мьютекса одним з потоків.

У Python використовують термін **lock** – замок.

Класи для реалізації блокування містяться у модулі threading.

Це класи Lock та RLock.

Класи Lock та RLock

Конструктори цих класів не потребують параметрів.

Обидва класи мають методи acquire – отримати блок - та release – звільнити блок.

Наприклад:

```
lock = Lock()
...
lock.acquire()
P
lock.release()
```

Класи Lock та Rlock.2

Блокування також можна виконувати з використанням менеджера контексту

```
with lock:
    P
```

Клас Lock це звичайний «замок», а RLock – «замок» з повторним входом (reentrant lock).

Для Lock після отримання блокування повторне отримання неможливе.

Спроба повторно отримати блокування до його звільнення призведе до «зависання» програми.

У таких випадках кажуть про глухий кут (deadlock).

Класи Lock та Rlock.3

Для RLock після отримання блокування повторне отримання можливе, але тільки з того ж потоку, з якого було отримане перше блокування.

На кожне отримання блокування має виконуватись своє звільнення блокування.

При цьому, для RLock, на відміну від Lock, це звільнення має виконатись у тому ж потоці, що й отримання блокування.

Приклад: додавання одиниць, версія 2

Розглянемо програму, що містить 2 потоки, які використовують одну і ту ж функцію для обчислення певного числа шляхом додавання одиниць.

Результат додавання будемо зберігати у змінній, що доступна обом потокам.

Використаємо блокування для коректного обчислення результату.

Події, умови, семафори, бар'єри

Окрім блокування, модуль threading містить інші примітиви для синхронізації потоків. Це події умови, семафори, бар'єри.

Подія – клас Event – слугує для інформування потоків про настання деякої події. Подію можна інтерпретувати як прапорець. Конструктор класу не має параметрів.

Події, умови, семафори, бар’єри.2

Клас містить методи:

Метод	Опис
is_set()	Чи встановлено подію
set()	Встановити подію. Усі потоки, що чекають на встановлення події, активізуються.
clear()	Очистити подію. Усі потоки, що чекають на встановлення події, блокуються.
wait(timeout=None)	Очікувати на встановлення події. Потік блокується до моменту встановлення події, або, якщо вказано timeout, - то максимум на timeout секунд.

Події, умови, семафори, бар’єри.3

Умови (Condition) схожі на події, але дозволяють вказати певну умову розблокування потоку або надіслати повідомлення про відновлення.

Семафори (Semaphore) використовують, коли є ресурс з обмеженою кількістю сутностей.

Семафор дозволяє паралельну роботу не більше заданої кількості потоків, які використовують вказаний ресурс.

Бар’єри (Barrier) призупиняють декілька потоків, поки вони не увійдуть до певного стану, після чого всі ці потоки розблокуються одночасно.

Таймери

Таймер (Timer) дозволяє стартувати потік через визначений інтервал у секундах.

Конструктор має вигляд:

```
Timer(interval, function, args=None, kwargs=None)
```

Щоб стартувати потік для функції fun без параметрів через 10 секунд, треба написати:

```
th = Timer(10, fun)
th.start()
```

Використання черг

Інколи для синхронізації потоків недостатньо просто використовувати блокування.

Класична проблема – це проблема постачальників – споживачів.

Потоки – постачальники постачають деякі дані, споживачі – обробляють ці дані.

Ці потоки можуть працювати з різною швидкістю і можлива ситуація, коли потоки споживачі ще не готові обробити дані, які надходять від постачальників.

Тоді ці дані треба десь тимчасово зберігати.

Для збереження та подальшої обробки даних використовують черги.

Використання черг.2

У Python черга, яку безпечно використовувати у багатопотокових програмах, реалізована у модулі queue.

У цьому модулі є декілька класів для черг, з яких ми розглянем головний – Queue.

Конструктор класу може мати 0 параметрів або 1 параметр – maxsize, що за угодою дорівнює 0.

Цей параметр, якщо не дорівнює 0, визначає максимально допустиму кількість елементів у черзі.

```
Queue(maxsize=0)
```

У модулі queue також описано класи виключень для черг: Empty – черга порожня, Full – черга заповнена максимальною кількістю елементів.

Основні методи класу Queue

Метод	Опис
put(item, timeout=None)	Додати item до черги. Якщо черга містить максимально можливу кількість елементів та timeout не встановлено, то черга буде очікувати на вільне місце. Якщо встановлено timeout та у черзі максимально можлива кількість елементів, то черга буде очікувати timeout секунд, щоб з’явилось вільне місце. Якщо після timeout секунд вільне місце не з’явилось, виникає виключення Full
get(timeout=None)	Взяти перший елемент з черги. Якщо черга порожня та timeout не встановлено, то черга буде очікувати на додавання елемента. Якщо встановлено timeout та черга порожня, то черга буде очікувати timeout секунд, щоб додався елемент. Якщо після timeout секунд елемент не додався, виникає виключення Empty

Приклад: черга показу результатів

Реалізувати правильний показ результатів багатьма потоками з використанням стандартної функції print

Черга показу результатів. Реалізація

Для розв’язання задачі опишемо клас PrintQueue

Цей клас організовує чергу повідомлень, які треба вивести, а також потік, що обробляє цю чергу та виводить повідомлення.

Поля класу:

self._queue - черга виведення

self._thread - потік, що обробляє чергу виведення

self._thread_stopped - подія, чи завершено роботу потоку

Методи класу: конструктор __init__, print – додати повідомлення до черги, stop – зупинити роботу потоку обробки черги.

Потік обробки черги реалізовано у методі _run

Пули потоків

Створювати новий потік у кожному випадку, коли починається паралельна обробка – не зовсім гарна ідея через накладні витрати, які необхідні для запуску нового потоку.

Способом обмежити кількість потоків, які виконуються одночасно, та спростити взаємодію з потоками є пули потоків.

У Python пул потоків реалізовано у класі ThreadPoolExecutor з модуля concurrent.futures.

Конструктор класу містить декілька параметрів, серед яких max_workers – максимальна кількість потоків у пулі.

```
executor = ThreadPoolExecutor(max_workers=None)
```

Пул потоків містить вбудовану чергу, з якої вибирає до виконання не більше заданої кількості потоків.

Пули потоків.2

Методи для об’єктів ThreadPoolExecutor

Метод	Опис
submit(fn, *args, **kwargs)	Додати функцію fn до пулу потоків. Повертає об’єкт класу Future
shutdown(wait=True)	Завершити роботу пулу потоків. Повертає управління одразу або після завершення роботи пулу у залежності від значення параметру wait
map(fn, *iterables)	Повертає послідовність результатів застосування функції fn до елементів iterables

Клас Future

Об’єкт класу Future – це майбутній виклик функції, що додається до пулу потоків.

Метод	Опис
result(timeout=None)	Повернути результат майбутнього виклику. Якщо вказано timeout та результат не обчислено після timeout секунд – ініціює виключення concurrent.futures.TimeoutError
cancel()	Відмінити виклик раніше доданої функції. Відмінити можна тільки виклик, який ще не почав виконуватись. Повертає успішність операції
cancelled() **running()** **done()**	Методи, що перевіряють стан майбутнього виклику. Повертають True, якщо відповідно: - виклик відмінено - виклик відмінено - виклик завершив виконання
add_done_callback(fn)	Додати функцію fn, яка буде викликатись по завершенні майбутнього виклику.

Приклад: побудова дерев каталогів та знаходження однакових файлів

Скласти програму, яка має графічний інтерфейс та забезпечує сканування вибраних дерев каталогів та знаходження у цих деревах однакових за змістом файлів.

Графічний інтерфейс має містити 2 списки: список доданих каталогів, що обробляються, та список каталогів, які вже оброблено.

Також мають бути кнопки «Додати…» (дерево каталогів), «Закрити» (завершити програму), «Зберегти» (зберегти дані про файли, у тому числі – однакові) для вибраних дерев каталогів.

Будемо вважати, що 2 файли мають однаковий зміст, якщо у них рівні розміри та контрольні суми.

Побудова дерев каталогів та знаходження однакових файлів. Реалізація

Оскільки у нас графічний інтерфейс, нам обов’язково знадобляться додаткові потоки. Інакше інтерфейс не буде відповідати, поки ми будемо обробляти чергове дерево каталогів.

Опишемо такі класи:

File – зберігає інформацію про файл та обчислює контрольну суму

DirTree – зберігає інформацію про дерево каталогів, сканує дерево каталогів у файловій системі

DirsForest – зберігає інформацію про додані дерева каталогів а також словник з усіма файлами, однакові файли зберігаються у одному елементі словника

DirsForestGui – реалізує графічний інтерфейс а також ініціює додавання дерева каталогів

Побудова дерев каталогів та знаходження однакових файлів. Реалізація.2

Для побудови використаємо два пули потоків: для сканування 1 дерева каталогів та для обчислення контрольної суми файлу.

Багатопроцесність

Розпаралелювання обчислень за допомогою декількох процесів, що виконуються одночасно, також може бути виконано у Python.

Засоби для цього зібрані у модулях:

- multiprocessing
- concurrent.futures
- subprocess

Модуль multiprocessing

Цей модуль переважно копіює функціональність модуля threading для потоків.

Головний клас – це клас Process. Конструктор класу виглядає так:

```
ps = Process(group=None, target=None,
     name=None, args=(), kwargs={})
```

Методи класу Process подібні до методів класу Thread

Модуль multiprocessing також містить замки, умови інші засоби синхронізації процесів.

У цьому ж модулі є клас для реалізації черг для процесів: Queue.

Приклад: виконання обчислення факторіалу у окремому процесі

Повторимо приклад з простим потоком та реалізуємо обчислення факторіалу у окремому процесі.

Пули процесів

По аналогії з пулами потоків, ми можемо створити пул процесів.

Засоби для створення такого пулу містяться у том ж модулі, що й для потоків, - concurrent.futures.

Відповідний клас називається ProcessPoolExecutor має конструктор та інші методи подібні до ThreadPoolExecutor.

По виконанні методу submit створюється новий майбутній виклик функції – об’єкт розглянутого раніше класу Future.

Приклад: пул процесів для обчислення чисел Фібоначчі

Використаємо пул процесів для обчислення різних чисел Фібоначчі за допомогою рекурсивної функції.

Результат будемо зберігати у вигляді списку кортежів: номер числа, значення.

Запуск підпроцесів

Підпроцеси – це окремі процеси з точки зору операційної системи. Для роботи з підпроцесами використовують модуль subprocess.

Стандартне використання цього модуля – запуск процесу, очікування виконання, обробка результату. Все це здійснює функція run

```
run(args, *, stdin=None, input=None,
         stdout=None, stderr=None)
```

args – це параметри – список, що включає назву процеса (шлях до файлу, що виконується), та параметри командного рядка, якщо є.

stdin, input, stdout, stderr дозволяють перенаправити введення/виведення нового процесу та зв’язати його з поточним процесом.

Приклад: запуск процесу для обчислення факторіалу, версії 1, 2

Запустимо наш попередній приклад з цієї теми – багатопотокову програму для обчислення факторіалу – як окремий процес.

Версія 1 просто запускає процес, версія 2 робить перенаправлення ввведення/виведенння.

Клас Popen

Більші можливості запустити новий процес та взаємодіяти з ним під час виконання надає клас Popen.

Запуск процесу здійснюється у конструкторі цього класу.

```
ps = Popen(args, stdin=None, stdout=None, stderr=None)
```

Конструктор має доволі багато параметрів, ми розглянем лише деякі.

args – це послідовність параметрів програми, які передаються у командному рядку. Першим параметром має бути назва (шлях до) програми, що буде виконуватись у вигляді окремого процесу.

stdin, stdout, stderr дозволяють перенаправити введення/виведення нового процесу.

Конструктор повертає об’єкт класу Popen та запускає процес на виконання. Запуск процесу не блокує програму, яка його запустила.

Клас Popen.2

Деякі методи Popen:

```
ps.wait(timeout=None)
```

очікує на завершення процесу, блокує програму до завершення або на timeout секунд.

```
ps.communicate(input=None, timeout=None)
```

комунікувати з процесом. Передати процесу вхідні дані, отримати виведення від процесу, очікувати на завершення процесу або timeout секунд.

Передати дані процесу можна за допомогою input, вказавши рядок байтів (або рядок символів).

Повертає кортеж з 2 файлів – stdout, stderr. Для того, щоб взаємодіяти з процесом за допомогою communicate, треба відповідним параметрам встановити значення PIPE, наприклад, stdout=PIPE. Аналогічно для stdin та stderr.

Клас Popen.3

```
ps.kill()
ps.terminate()
```

завершити виконання процесу.

ps.kill() – більш «м'який» варіант завершення (доступне у Linux, OS X).

Приклад: запуск процесу для обчислення факторіалу, версії 3, 4

Запустимо наш попередній приклад з цієї теми – багатопотокову програму для обчислення факторіалу – як окремий процес.

Використаємо Popen.

Версія 3 взаємодіє з процесом за допомогою PIPE, версія 4 передає вхідні дані з окремого файлу.

Асинхронне програмування

Рух за зменшення кількості потоків у Python призвів до того, що з'явились бібліотеки, які зменшують кількість необхідних потоків до 1.

Ідея цих бібліотек вперше була реалізована у twisted.

Вона базується на циклі обробки подій, що дещо нагадує функціонування графічного інтерфейсу.

Кожна функція, що виконується у такому середовищі, має або швидко закінчувати роботу, або час від часу передавати управління циклу обробки подій.

Такий стиль програмування називають асинхронним програмуванням.

Асинхронне програмування.2

У сучасних версіях Python асинхронне програмування підтримує пакет asyncio.

У мову Python було внесено навіть декілька нових ключових слів: async def, async for, await тощо.

Асинхронне програмування є доволі популярним та дозволяє розв'язувати певний клас задач введення/виведення, що раніше розв'язувались за допомогою потоків.

Слід зазначити, що асинхронне програмування не відміняє багатопотоковість, а радше доповнює її.

Резюме

Ми розглянули:

1. Процеси та потоки
2. Багатопотоковість у Python. GIL
3. Клас Thread. Потоки–«демони»
4. Використання потоками спільних даних. Ситуація змагальності (race condition)
5. Синхронізація потоків. Класи Lock та RLock
6. Події, умови, семафори, бар’єри. Таймери
7. Використання черг
8. Багатопроцесність. Модуль multiprocessing
9. Пули процесів
10. Запуск підпроцесів. Клас Popen

Де прочитати

1. Уэсли Дж. Чан. - Python: создание приложений. Библиотека профессионала, 3-е изд. Пер.с англ. - М. : ООО "И.Д. Вильяме", 2015. - 816
2. Mark Lutz - Programming Python. 4th Edition - 2011
3. Peter Norton, Alex Samuel, David Aitel та інші - Beginning Python
4. Magnus Lie Hetland - Beginning Python from Novice to Professional, 2nd ed – 2008
5. Куан Нгуен. Полное руководство параллельного программирования на Python. - 2018 Packt Publishing. - http://onreader.mdl.ru/MasteringConcurrencyInPython/content/Index.html
6. Прохоренок Н.А. - Python 3 и PyQt. Разработка приложений – 2012
7. Mark Pilgrim - Dive into Python, Version 5.4 - 2004
8. Noah Gift, Jeremy M. Jones Python for Unix and Linux System Administration
9. https://realpython.com/intro-to-python-threading/#daemon-threads