Лекція 14: Конкурентність та паралелізм

Окрім збільшення швидкості роботи комп'ютерних процесорів існує й інший підхід: використання кількох процесорних ядер. На відміну від одноядерних процесорів, багатоядерні процесори мають 2 або більше обчислювальних ядер, що можуть виконувати обчислення одночасно.

Зазвичай програми виконуються тільки на одному ядрі, виконуючи код послідовно, інструкція за інструкцією. Однак часто послідовні інструкції не обов'язково мають бути послідовними, порядок їх виконання не суттєвим, тому, виконуючи їх паралельно на різних ядрах процесора можна суттєво пришвидшити роботу програми.

Для роботи з багатоядерними процесорами використовується абстракція потоків виконання (англ. execution threads). Кілька потоків виконання можуть виконуватись одночасно на різних ядрах процесора, або на одному ядрі процесора, коли процесор постійно переключається між виконанням різних потоків, створюючи ілюзію одночасного виконання.

В сучасних операційних системах кожен процес має хоча б один основний потік виконання. Оскільки кожен процес має окремий потік виконання, операційні системи можуть виконувати кожен з процесів на іншому ядрі процесора таким чином використовуючи переваги багатоядерних процесорів навіть якщо кожен з процесів виконується тільки в одному потоці.

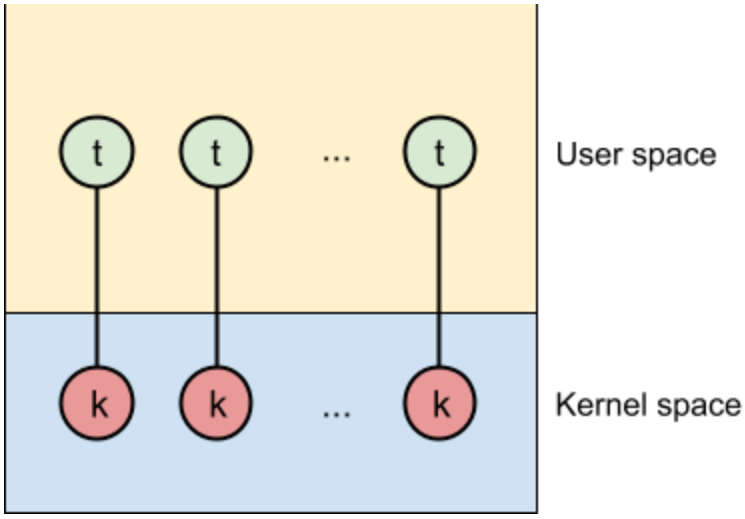
Потоки поділяють на потоки ядра (англ. kernel threads, kernel-level threads) та потоки користувацького простору (англ. user-space threads, user-level threads). Потоки ядра є низькорівневим представленям потоків реалізованим на рівні ядра операційної системи. Потоки користувацького простору є абстракцією поверх потоків ядра та реалізовані у середовищі виконання (інтерпретаторі чи віртуальній машині) чи окремих бібліотеках (у випадку з мовами програмування, що компілюються у машинний код). Потоки користувацького простору дозволяють реалізувати підтримку багатопоточності в операційних системах, що не підтримують багатопоточність, однак, оскільки усі сучасні операційні системи підтримують багатопоточність, то потоки користувацького простору використовуються виключно як рівень абстракції поверх потоків ядра [*] [**].

* Implementing threads :: Operating systems 2018 (http://www.it.uu.se/education/course/homepage/os/vt18/module-4/implementing-threads/)

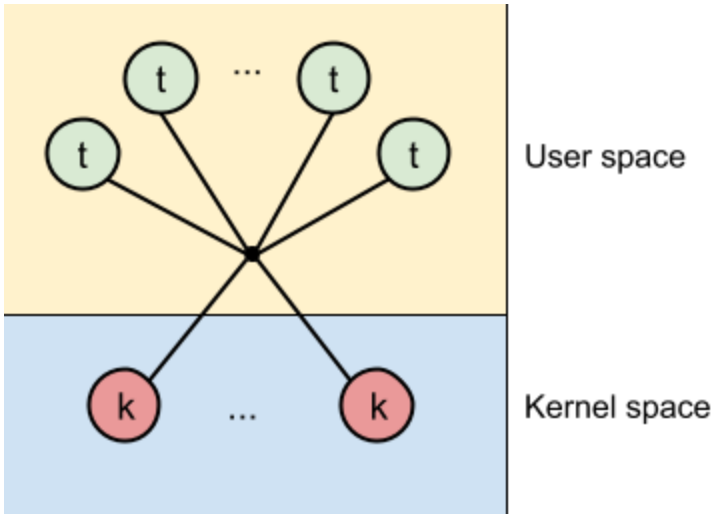
** Operating Systems: Threads (https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/4_Threads.html)

Потік користувацького простору не обов’язково відповідає потоку ядра. Існує ряд моделей того як потоки користувацького простору співвідносяться з потоками ядра.

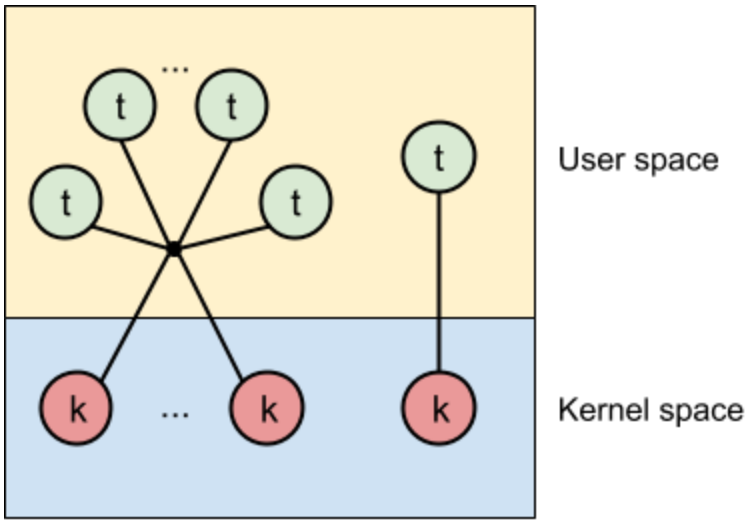
Модель один до одного — для кожного потоку користувацького простору створюється відповідний потік ядра.



Модель багато до багатьох — для N-ої кількості потоків користувацького простору створюється M потоків ядра. В даному випадку в різні моменти часу деякі потоки користувацького простору виконуються на окремих потоках ядра, а деякі ділять один потік ядра виконуючись почергово.



Дворівнева модель — поєднання моделі один до одного та багато до багатьох. Дана модель схожа до моделі багато до багатьох, однак дозволяє деяким потокам користувацького простору мати окремо виділені потоки ядра як у потоках ядра.



Схожим чином потоки ядра операційної системи виконуються на окремих ядрах процесора. При наявності багатьох потоків виконання деякі потоки виконуються паралельно на окремих ядрах, а деякі ділять для виконання одне ядро процесора. Дана особливість відображає дві характеристики виконання потоків: паралелізм та конкурентність.

Паралелізм — це можливість виконувати потоки одночасно, на різних ядрах процесора.

Конкурентність — це можливість виконувати потоки почергово розділяючи одне ядро процесора, створюючи ілюзію одночасного виконання.

Багатопоточність в Python. Пакет threading [*]

* threading — Thread-based parallelism — Python 3.8.0 documentation (https://docs.python.org/3/library/threading.html)

Стандартна бібліотека Python містить пакет threading для роботи з багатопоточністю. Даний пакет містить ряд класів для створення потоків, їх синхронізації та багато іншого.

Розглянемо клас “threading.Thread”, що дозволяє створювати потоки. Даний клас дозволяє створювати окремі потоки виконання, де кожен екземпляр даного класу відповідає окремому потоку виконання.

Існують два способи використання класу “threading.Thread”. Можна створити його екземпляр, передавши аргументом “target” функцію, що має бути виконана в окремому потоці, або створивши окремий клас, що наслідує “threading.Thread” та перевизначає метод “run”. Нижче наведені приклади використання кожного зі способів.

main.py	python

```python
from threading import Thread

def function_to_execute_in_thread():
    ...

thread = Thread(target=function_to_execute_in_thread)
```

main.py python

```python
from threading import Thread

class CustomThread(Thread):
    def run(self):
        ...
```

Методи start та join

Екземпляри “Thread” мають методи “start” та “join” для початку та закінчення роботи потоку відповідно. При виклику методу “start” відповідний потік починає виконуватись. Важливо, що код, що знаходиться після виклику методу “start” продовжує виконуватись в основному потоці. Метод “join” дозволяє чекати на закінчення роботи відповідного потоку. Код, що знаходиться після виклику “join” не буде виконаним поки відповідний потік не закінчить своє виконання. Використаємо наступний код для демонстрації роботи даних методів, використовуючи вивід в термінал для визначення того в якому порядку виконується код в різних потоках.

main.py python

```python
from threading import Thread

def function_to_execute_in_thread():
    print(f"CUSTOM THREAD: print 1")
    print(f"CUSTOM THREAD: print 2")

thread = Thread(target=function_to_execute_in_thread)

print("MAIN THREAD: Before start")
thread.start()
print("MAIN THREAD: After start")
thread.join()
print("MAIN THREAD: After join")
```

Виконаємо дану програму:

terminal

```
$ python main.py
MAIN THREAD: Before start
CUSTOM THREAD: print 1
CUSTOM THREAD: print 2
MAIN THREAD: After start
MAIN THREAD: After join
```

В даному випадку обидва вирази в окремому потоці були виконані перед кодом, після виклику методу “start”. Такий порядок виконання аналогічний простому послідовному виклику функції. Спробуємо запустити програму знову:

terminal

```
$ python main.py
MAIN THREAD: Before start
CUSTOM THREAD: print 1
MAIN THREAD: After start
CUSTOM THREAD: print 2
MAIN THREAD: After join
```

При повторному виконанні другий вираз в окремому потоці був виконаний після коду, що знаходиться після виклику методу “start”. Оскільки код виконується в різних потоках, то порядок виводу може змінюватись від запуску до запуску програми.

Порівняння тривалості виконання послідовного виконання та виконання в окремих потоках

Використаємо клас “Thread” щоб порівняти тривалість піднесення у степінь N-ої кількості чисел. Для визначення тривалості виконання використаємо пакет “timeit” зі стандартної бібліотеки Python.

Спочатку реалізуємо програму, що підносить у степінь задану кількість чисел послідовно, без використання потоків.

main.py python

```python
import sys
from timeit import timeit

REPEAT_NUMBER = int(sys.argv[1])
```

```python
def main():
    sqrts = []
    for num in range(REPEAT_NUMBER):
        sqrts.append(pow(num, 2))
    return sqrts

print(timeit(main, number=1))
```

Дана програма приймає аргумент з кількістю чисел, що потрібно піднести у степінь та підносить у степінь відповідну кількість чисел та виводить тривалість виконання в секундах. Виконаємо дану програму, піднявши у степінь 100000 чисел.

terminal

```
$ python main.py 100000
0.05744136101566255
```

При послідовному виконанні підняття у степінь ста тисяч чисел зайняло трохи більше 57 мікросекунд. Тепер створимо аналогічну програму, що виконує те ж саме, тільки кожне підняття у степінь буде відбуватись в окремому потоці.

main.py

```python
import sys
from functools import partial
from threading import Thread
from timeit import timeit

REPEAT_NUMBER = int(sys.argv[1])

def main():
    threads = []
    for num in range(REPEAT_NUMBER):
        t = Thread(target=partial(pow, num, 2))
        t.start()
        threads.append(t)
    for thread in threads:
        thread.join()

print(timeit(main, number=1))
```

Виконаємо дану програму, піднявши у степінь такі ж 100000 чисел.

terminal

```
$ python main.py 100000
8.182956686941907
```

Програма, що виконує такі ж сто тисяч операцій в окремих потоках виконується більше восьми секунд на комп'ютері з багатоядерним процесором. Здається, що такий результат не має жодного сенсу, однак він спричинений однією з особливостей потоків у стандартній реалізації інтерпритатора Python (CPython), що буде розгляненим у наступному розділі.

GIL (Global interpreter lock) [*]

* GlobalInterpreterLock - Python Wiki (https://wiki.python.org/moin/GlobalInterpreterLock)

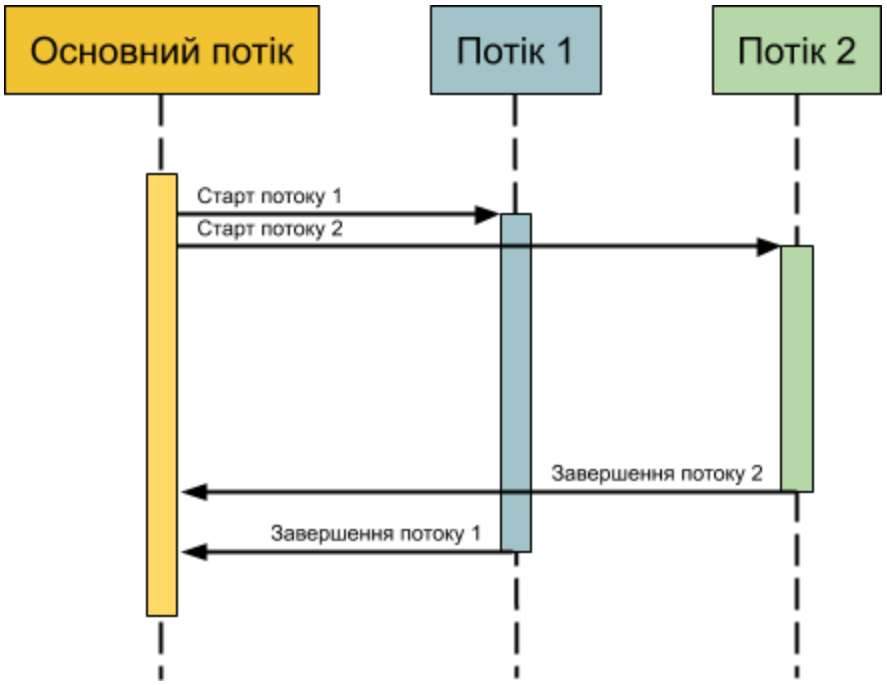
У стандартній реалізації інтерпретатора Python (CPython) global interpreter lock або GIL — це механізм, що захищає доступ до Python об'єктів не дозволяючи потокам виконувати Python байт-код одночасно, що робить потоки в CPython не паралельними. Основною причиною даного обмеження є те, що управління пам'яттю в Python не розраховане на одночасне виконання в кількох потоках.

Варто розуміти, що GIL не є обов'язковою частиною інтерпретатора Python, та існують реалізації інтерпретатора, що не мають цього обмеження та дозволяють виконувати потоки паралельно на різних ядрах процесора. Серед версій інтерпрератора Python, що не мають GIL, є Jython (використовує віртуальну машину Java як середовище виконання) та IronPython (використовує .NET CLR як середовище виконання), також в окремих випадках Cython (не плутати з CPython) дозволяє відключати GIL.

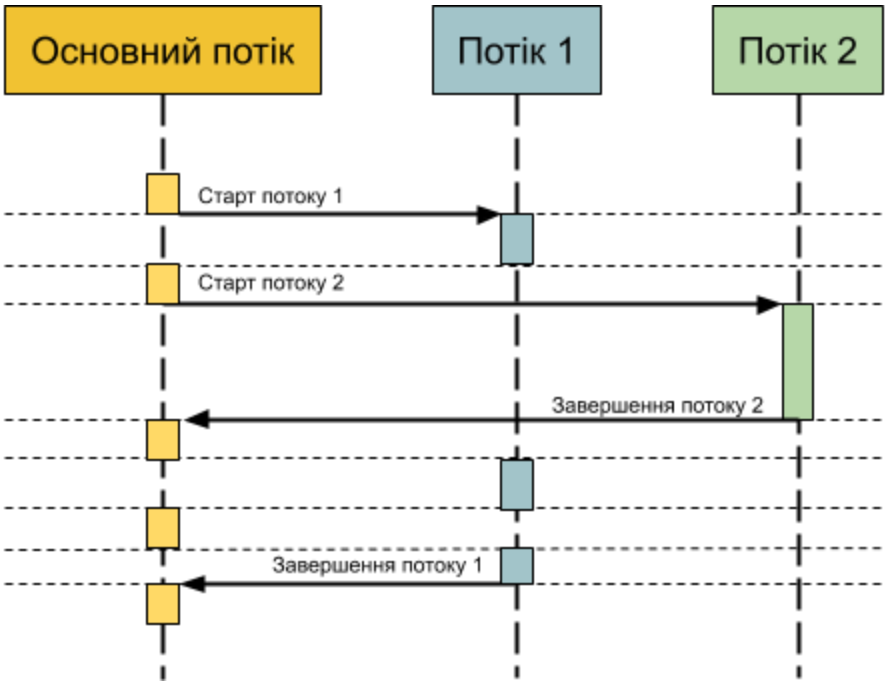
Було багато спроб прибрати GIL зі стандартної реалізації Python, однак жодна з них не була успішною оскільки до будь-якої спроби прибрати GIL є ряд вимог: відсутність надмірного ускладнення інтерпретатора, ідентична чи більша швидкість виконання багатопоточних програм при типовому для Python використанні (більше деталей дані), відсутність зменшення швидкості виконання однопоточних програм, зворотна сумісність з вже наявним в інтерпретаторі функціоналом.

Наявність GIL означає, що потоки в CPython не можуть виконуватись паралельно, що не дозволяє багатопоточним Python програмам використовувати переваги багатоядерних процесорів. З першого погляду може здатись, що такі потоки не мають жодного сенсу, однак це не так. Хоча виконання потоків в CPython не є паралельним, однак воно є конкурентним. В прикладі раніше з використанням інструкції "print" було показано, що виконання функції в окремому потоці не еквівалентне простому виклику цієї ж функції. Код в окремих потоках виконується почергово.

Паралельне виконання потоків продемонстровано наступною діаграмою.



Конкурентне виконання потоків продемонстровано наступною діаграмою. В даному випадку потоки не виконуються одночасно, натомість цілі потоки або їх частини виконуються почергово (включно з основним потоком).



В CPython потоки переключаються не випадково. Переключення між потоками відбувається в моменти, коли потік взаємодіє з вводом/виводом (англ. input/output, IO). У

прикладі вище з використанням інструкцій “print” такою взаємодією з вводом/виводом був вивід у потік стандартного виводу (вивід тексту в термінал). Серед таких операцій вводу/виводу можна виділити наступні:

- Взаємодія зі стандартними потоками вводу/виводу (вивід у термінал, чи читання значень, введених користувачем у термінал);
- Робота з файлами (запис чи читання цілого файлу чи його частин);
- Робота з мережею (відправка HTTP запитів);
- Виклик функції “time.sleep”.

Важливо розуміти, що поки в потоці виконується код без операцій вводу/виводу інтерпретатор блокується даним потоком та не буде переключатись на виконання інших потоків. Це означає, що потоки, які не виконують жодних операцій з вводом/виводом, будуть виконані від початку до кінця без переключення на інші потоки роблячи виконання таких потоків неконкурентним та по порядку виконання коду ідентичним до простого виклику функції. Таких потоків в CPython варто уникати та у випадках, коли необхідно виконати обчислення без операцій вводу/виводу паралельно, використовувати окремі процеси або розширення інтерпретатора, що не обмежені GIL-ом, оскільки реалізовані на C (наприклад numpy).

Наявність GIL пояснює чого у прикладі з порівнянням швидкості роботи при використанні потоків швидкість роботи програми не була збільшена, однак GIL не пояснює чому тривалість виконання була набагато більшою за тривалість виконання при послідовному виконанні. Причиною набагато більшої тривалості виконання є те, що переключення між потоками відбувається не миттєво, тому в такому прикладі переключення між потоками займає більшу частину часу.

Конкурентні потоки

Продемонструємо ситуацію, коли використання потоків в CPython може дійсно зменшити загальну тривалість виконання. Одним з типових сценаріїв на якому можна це продемонструвати є виконання запитів по мережі для яких порядок виконання не є важливим.

Спочатку реалізуємо версію програми, що виконує запити послідовно. Запити будемо робити до “httpbin.org/delay/1” використовуючи бібліотеку “requests”. Кожен запит триває одну секунду плюс мережеві затримки. Реалізація такої програми наведена нижче:

main.py — python

```python
import sys
from timeit import timeit
import requests
```

```
REPEAT_NUMBER = int(sys.argv[1])

def main():
    for _ in range(REPEAT_NUMBER):
        make_request()

def make_request():
    requests.get('http://httpbin.org/delay/1').json()

print(timeit(main, number=1))
```

Виконаємо дану програму, щоб виконати 5 запитів послідовно:

terminal

```
$ python main.py 5
6.5989407890010625
```

Послідовно 5 запитів були виконані за 6.5 секунд. Тепер спробуємо використати потоки щоб виконувати дані запити конкурентно в окремих потоках. Код даної програми наведений нижче:

main.py python

```
import sys
from threading import Thread
from timeit import timeit
import requests

REPEAT_NUMBER = int(sys.argv[1])

def main():
    threads = []
    for num in range(REPEAT_NUMBER):
        t = Thread(target=make_request)
        t.start()
        threads.append(t)
    for thread in threads:
        thread.join()

def make_request():
    requests.get('http://httpbin.org/delay/1').json()

print(timeit(main, number=1))
```

Виконаємо дану програму, щоб виконати 5 запитів конкурентно:

terminal

```
$ python main.py 5
1.367698780959472
```

Конкурентно 5 запитів були виконані за 1.3 секунди. В даному випадку використання потоків допомагає значно пришвидшити роботу програми, оскільки суттєва частина тривалості виконання коду, що виконується в потоці, припадає на операції вводу/виводу, в даному випадку — виконання запиту по мережі.

При оцінці необхідності використання потоків в Python використовуються такі терміни як CPU-bound та I/O-bound.

CPU-bound — це характеристика, що визначає код, тривалість виконання якого визначається тривалістю виконання необхідних операцій процесором. В Python CPU-bound код — це код, що не містить операцій вводу-виводу (наприклад, обчислення чи просто логіка програми). Варто розуміти, що CPU-bound код хоч може але не обов'язково має якось суттєво навантажувати процесор.

I/O-bound — це характеристика, що визначає код, тривалість виконання якого в основному визначається чеканням на операції вводу-виводу, тобто більша частина тривалості виконання програми припадає не на безпосереднє виконання, а на чекання на відповіді на запити по мережі, читання чи запис файлів або делегування завдань окремим процесам.

На практиці різні частини програм можуть бути в тій чи іншій мірі CPU-bound або I/O-bound. Відповідно до того чи код програми чи її частини більше CPU-bound чи I/O-bound приймаються рішення щодо того чи доцільно використовувати потоки для пришвидшення роботи програми. Для більш CPU-bound програм потоки можуть не дати жодного ефекту на тривалість виконання тільки ускладнюючи читання коду або навіть збільшувати загальну тривалість. Для більш I/O-bound програм потоки можуть суттєво зменшити загальну тривалість виконання програми.

Одним з типових застосувань Python є програмування серверної частини веб-сайтів. Такий код є здебільшого I/O-bound, виконуючи запити до сторонніх сервісів, працюючи з базою даних чи файловою системою, тому використання потоків як правило дозволяє пришвидшити їх роботу. Як правило окремі запити від клієнтів виконуються в окремих потоках, що дозволяє швидше виконувати конкурентні запити до сервера, виконуючи код для нового запиту поки процес обробки старого запиту чекає на завершення операції вводу-виводу. Таке розділення виконання запитів на окремі потоки може виконуватись самою програмою або сторонніми інструментами типу WSGI-серверів (прошарок між Python програмою та веб-сервером, наприклад, Nginx чи Apache), наприклад Gunicorn чи uWSGI.

Іншим типовим застосуванням Python є статистика, аналітика даних та машинне навчання. Такий код є здебільшого CPU-bound, тому використання CPython як правило не дасть ніяких позитивних ефектів. Натомість використовуються бібліотеки, що містять розширення інтерпретатора на низькорівневій мові типу C, що дозволяють виконувати статистичні операції набагато швидше та, навіть, виконувати їх на низькорівневих дійсно паралельних потоках. Таким чином швидкодія обробки даних низькорівневими розширеннями поєднується з виразністю та відносною легкістю навчання характерними для Python.

ThreadPool та ThreadPoolExecutor

Використовуючи threading.Thread можна створювати та запускати потоки, однак, як було продемонстровано раніше, переключення між потоками займає час, тому конкурентне виконання великої кількості функцій може бути не надто ефективним. Вирішити дану проблему можна поєднавши конкурентне та послідовне виконання, використавши паттерн пул потоків (англ. Thread pool).

Замість того, щоб виконувати кожну функцію у своєму потоці, кількість потоків є обмеженою розміром пулу. Розмір пулу визначає яка максимальна кількість потоків може виконуватись в конкретний момент часу. Якщо потрібно виконати більшу кількість операцій за розмір пулу, то частина операцій буде розподілена по потокам, а частина буде чекати на вільні потоки та, як тільки такі знайдуться, відповідні потоки почнуть виконувати наступні операції.

В Python є дві реалізації пулу потоків: multiprocessing.pool.ThreadPool [*] та concurrent.futures.ThreadPoolExecutor [**] (не доступний для Python 2).

* 16.6. multiprocessing — Process-based “threading” interface — Python 2.7.17 documentation (https://docs.python.org/2/library/multiprocessing.html) (сам multiprocessing.pool.ThreadPool не задокументований)

** concurrent.futures — Launching parallel tasks — Python 3.8.0 documentation (https://docs.python.org/3/library/concurrent.futures.html#concurrent.futures.ThreadPoolExecutor)

multiprocessing.pool.ThreadPool

Клас ThreadPool приймає розмір пулу при створенні екземпляру та дозволяє конкурентно виконуват функції в пулі використовуючи метод “map”. Метод “ThreadPool.map” працює схоже до вбудованої функції “map”, приймаючи першим аргументом функцію, а другим — аргументи для цієї функції. По завершенню використання екземпляру ThreadPool

Використаємо ThreadPool для програми, що робить запити до веб-сайту. Нижче наведений код програми:

main.py python

```python
import sys
from multiprocessing.pool import ThreadPool
from timeit import timeit
import requests

REPEAT_NUMBER = int(sys.argv[1])
NUMBER_OF_WORKERS = 3

def main():
    pool = ThreadPool(NUMBER_OF_WORKERS)

    results = pool.map(make_request, REPEAT_NUMBER *
['http://httpbin.org/delay/1'])
    print(f'Number of results: {len(results)}')

    pool.terminate()
    pool.join()

def make_request(url):
    return requests.get(url).json()

print(timeit(main, number=1))
```

В даному прикладі розмір пулу визначається змінною “NUMBER_OF_WORKERS” та рівний трьом. Спробуємо виконати дану програму з різною кількістю запитів, що потрібно виконати.

terminal

```
$ python main.py 1
Number of results: 1
1.4460977921262383
$ python main.py 3
Number of results: 3
1.3436655059922487
$ python main.py 4
Number of results: 4
2.689666740130633
```

З виводу кожного з виконань програми бачимо, що при кількості конкурентних завдань меншій або рівній розміру пулу загальна тривалість виконання практично не відрізняється,

оскільки самі завдання не блокують інтерпретатор та виконуються кожен у своєму потоці. Однак якщо кількість конкурентних завдань більша за розмір пулу, в даному випадку 4, частина завдань має чекати коли з'являться вільні потоки.

Більше інформації про методи класу ThreadPool можна дізнатись в офіційній документації класу "multiprocessing.Pool", оскільки на даний момент офіційна документація по саме ThreadPool відсутня.

concurrent.futures.ThreadPoolExecutor

Клас "ThreadPoolExecutor" також приймає розмір пулу при створенні екземпляру пулу. На відміну від класу "multiprocessing.pool.ThreadPool" офіційна документація "ThreadPoolExecutor" рекомендує використовувати його як контекст-менеджер.

ThreadPoolExecutor.map

"ThreadPoolExecutor" також має метод "map", що працює аналогічним чином як і у "multiprocessing.pool.ThreadPool", за виключенням того, що повертається генератор з результатами, а не список. Нижче наведений приклад використання методу "ThreadPoolExecutor.map".

map.py python

```python
import sys
from concurrent.futures import ThreadPoolExecutor
from timeit import timeit
import requests

REPEAT_NUMBER = int(sys.argv[1])
NUMBER_OF_WORKERS = 3

def main():
    pool = ThreadPoolExecutor(max_workers=NUMBER_OF_WORKERS)
    with pool:
        results = pool.map(make_request, REPEAT_NUMBER *
['http://httpbin.org/delay/1'])
        print(f'Results {type(results)}. Number of results:
{len(list(results))}')

def make_request(url):
    return requests.get(url).json()

print(timeit(main, number=1))
```

Виконаємо дану програму:

terminal

```
$ python map.py 3
Results <class 'generator'>. Number of results: 3
1.3086274662055075
```

ThreadPoolExecutor.submit

Окрім методу “map” “ThreadPoolExecutor” має метод “submit”, що є набагато гнучкішим. Метод “ThreadPoolExecutor.submit” на відміну від “map” дозволяє виконувати в пулі різні функції з різною кількістю аргументів.

Метод “ThreadPoolExecutor.submit” дозволяє вказати одну функцію та аргументи для неї, що потрібно виконати у пулі. Відповідно, для виконання кількох функції необхідно виконати метод “submit” для кожної з них.

Значення, що повертає метод “submit” не є результатом виконання функції, що була передана. Натомість це так званий future-об’єкт (concurrent.futures.Future) [*].

* class concurrent.futures.Future (https://docs.python.org/3/library/concurrent.futures.html#concurrent.futures.Future)

concurrent.futures.Future

Для пояснення future-об’єктів можна провести аналогією з номерками замовлень в піцерії. Зробивши замовлення (виконавши “submit” для виконання функції в пулі) покупець одразу отримує номерок (future-об’єкт). Даний номерок не є замовленням, але з ним можна отримати замовлення коли воно буде готовим. Так само future-об’єкт не є результатом, але за допомогою нього можна отримати результат коли він буде готовим (коли відповідна функція завершить своє виконання в пулі).

Спробуємо використати Future-об’єкт сам по собі, без “ThreadPoolExecutor”.

Python REPL

```
>>> from concurrent.futures import Future
>>> f1 = Future()
>>> f1.set_result('My result')
>>> f1.result()
'My result'
>>> f2 = Future()
>>> f2.set_exception(Exception('My error'))
>>> f2.result()
Traceback (most recent call last):
  ...
```

В першому випадку методом “set_result” ми задаємо результат, що має повернути future-об’єкт та дістаємо цей результат викликом методу “result”. В другому випадку ми задаємо помилку методом “set_exception” та маємо цю помилку викинутою при виклику методу “result”. В третьому випадку ми не задаємо результатуу чи помилк, а, натомість, просто викликаємо метод “result” задавши аргумент “timeout” у дві секунди та отримуємо “TimeoutError” через 2 секунди, оскільки результат був відсутній (без аргументу timeout чекання на результат продовжувалося б безкінечно).

Використання ThreadPoolExecutor.submit

Тепер використаємо метод “ThreadPoolExecutor.submit” на практиці.

submit.py python

```python
import sys
from concurrent.futures import ThreadPoolExecutor
from timeit import timeit
import requests

REPEAT_NUMBER = int(sys.argv[1])
NUMBER_OF_WORKERS = 3

def main():
    pool = ThreadPoolExecutor(max_workers=NUMBER_OF_WORKERS)
    with pool:
        results_futures = []
        for _ in range(REPEAT_NUMBER):
            results_futures.append(pool.submit(make_request, delay=1))
        results = [fut.result() for fut in results_futures]
        print(f'Number of results: {len(results)}')

def make_request(delay):
    return requests.get(f'http://httpbin.org/delay/{delay}').json()

print(timeit(main, number=1))
```

В даному прикладі викликаючи метод “submit” програма зберігає future-об’єкти у список, а потім з цих future-об’єктів один за одним дістаються результати.

Щодо використання екземплярів “ThreadPoolExecutor” як контекст-менеджерів варто зазначити, що потоки з пулу видаляються при виході з контексту (по завершенню тіла блоку with), тому метод “submit” можна викликати виключно всередині контексту. Тоді як результати у future-об’єктах доступні і поза межами контексту. Якщо результат дістається з future-об’єкта всередині контексту, то ми будемо чекати коли відповідна функція в пулі

буде виконаною за збереже результат у future-об’єкт. Якщо результат дістається з future-об’єкта поза межами контексту, то результат дістається миттєво, оскільки при виході з контексту пул чекає на завершення всіх функцій перед тим як видалити потоки в пулі. Нижче наведений приклад використання future-об’єкту та методу “submit” поза межами контексту.

Python REPL

```
>>> from concurrent.futures import ThreadPoolExecutor
>>> with pool:
...     f = pool.submit(math.pow, 2, 3)
>>> print(f.result())
8.0
>>> f2 = pool.submit(math.pow, 2, 4)
Traceback (most recent call last):
  ...
RuntimeError: cannot schedule new futures after shutdown
```

Висновки

Використання пулу потоків є рекомендованим при конкурентному виконанні невизначеної кількості операцій, щоб уникнути створення величезної кількості потоків коли таких операцій є дуже багато. Однак, часто пули потоків використовують і для невеликої фіксованої кількості операцій, що необхідно виконати в потоках, через зручні програмні інтерфейси.

Багатопроцесність в Python. Пакет multiprocessing [*]

* multiprocessing — Process-based parallelism — Python 3.8.0 documentation (https://docs.python.org/3/library/multiprocessing.html)

Не зважаючи на неможливість паралельного виконання потоків в CPyhton, оптимізувати саме обчислення через паралельне виконання таки можливо. Для цього використовується так звана багатопроцесність (англ. multiprocessing). В Python для створення процесів з Python програм існують два пакети зі стандартної бібліотеки: subprocess [**] та multiprocessing [*]. Пакет subprocess більш загальний та орієнтований на запуск з Python програм будь-який процесів та комунікацію з ними через передачу довільних даних стандартними потоками вводу-виводу. Пакет multiprocessing орієнтований саме на створення дочірніх Python процесів, спрощуючи створення дочірніх процесів та комунікацію між основним та дочірніми процесами. В даному розділі увагу буде зосереджено саме на пакеті “multiprocessing”.

** subprocess — Subprocess management — Python 3.8.0 documentation (https://docs.python.org/3/library/subprocess.html)

Клас Process [*]

* The Process class (https://docs.python.org/3/library/multiprocessing.html#the-process-class)

Клас “multiprocessing.Process” схожий на “threading.Thread”, він має такі ж аргументи конструктора та методи для роботи з ним але замість потоків створює дочірні Python процеси. Продемонструємо роботу даного класу на прикладі.

main.py python

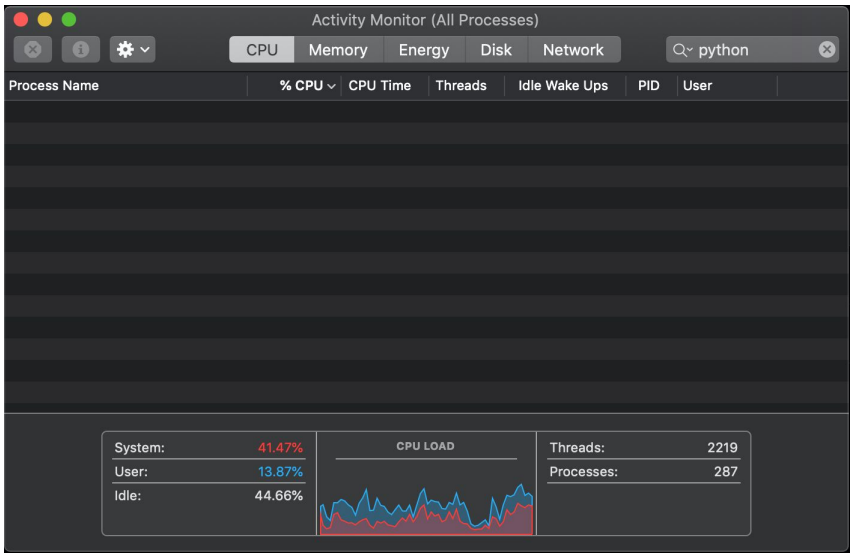
```python
from multiprocessing import Process
from time import sleep

def function_to_execute_in_child_process():
    sleep(10)

sleep(10)
process = Process(target=function_to_execute_in_child_process)
process.start()
process.join()
```

Дана програма при запуску чекає 10 секунд, після чого створює дочірній процес, що чекає 10 секунд, після чого обидва процеси завершуються. Очікування додані для того, щоб можна було побачити процеси в диспетчері задач чи іншій програмі, що показує процеси в операційній системі.

Перед запуском даної програми відкриємо диспетчер задач та обмежимо процеси, що відображаються, до тільки Python процесів за допомогою пошуку.

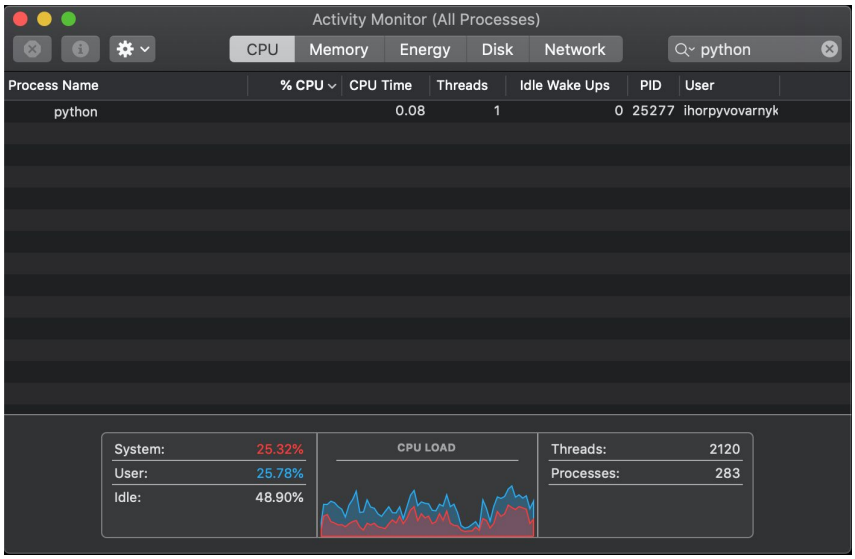


Тепер запустимо дану програму наступною командою:

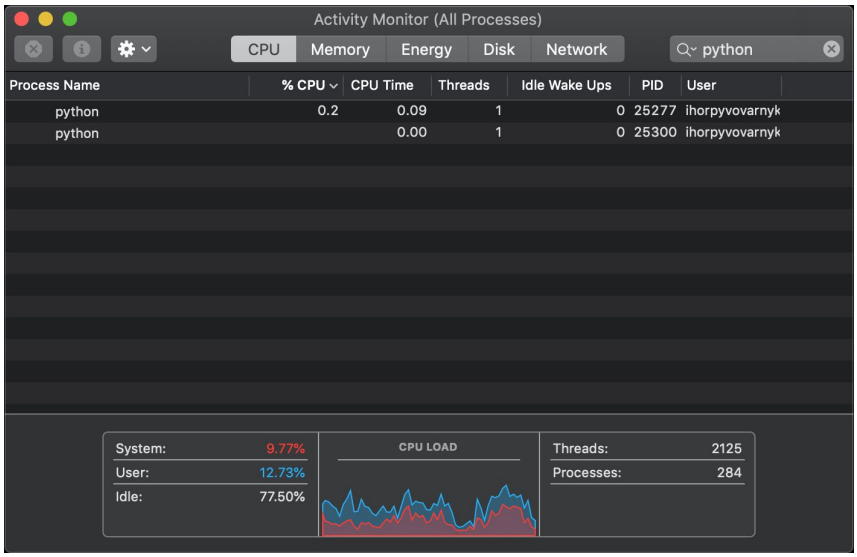
terminal

```
$ python main.py
```

В диспетчері задач бачимо, що з’явився Python процес.



Через 10 секунд бачимо, що з’явився і другий (дочірній) процес.



Згадаємо раніше використану програму з послідовного виконання CPU-bound операцій.

main.py python

```python
import sys
from timeit import timeit

REPEAT_NUMBER = int(sys.argv[1])

def main():
    sqrts = []
    for num in range(REPEAT_NUMBER):
        sqrts.append(pow(num, 2))
    return sqrts

print(timeit(main, number=1))
```

Використаємо дану програму для порівняння з версією, що використовує багатопроцесність. Спочатку запустимо дану програму з, наприклад тисячою повторень.

terminal

```
$ python main.py 1000
0.0005852999165654182
```

Бачимо, що дана програма виконалась за долю секунди. Тепер створимо версію, що виконує кожну з операцій в окремому процесі.

main.py python

```python
import sys
from multiprocessing import Process
from timeit import timeit

REPEAT_NUMBER = int(sys.argv[1])

def main():
    processes = []
    for num in range(REPEAT_NUMBER):
        p = Process(target=pow, args=(num, 2))
        p.start()
        processes.append(p)
    for process in processes:
        process.join()

print(timeit(main, number=1))
```

Запустимо дану програму з тією ж самою кількістю повторень:

terminal

```
$ python main.py 1000
2.1284127752296627
```

Бачимо, що на виконання такої програму потрібно суттєво більше часу. Причина в тому, що процеси створюються не миттєво. Більше того, створення процесу займає більше часу ніж створення потоку, те саме стосується і використання ресурсів процесора та оперативної пам'яті. До цього додається ще й час, що витрачається на старт Python-інтерпретатора в дочірньому процесі. В нашому прикладі ми створюємо велику кількість процесів, що виконують просту операцію, тому для кожного з процесів більшу частину часу займає не виконання цієї операції, а старт самого процесу та інтерпретатора.

Для того, щоб дійсно пришвидшити програму, необхідно щоб кожен з процесів виконував більше корисної роботи і відповідно старт процесу та інтерпретатора не займав би суттєву кількість часу.

Змінимо програму таким чином, зменшити кількість використовуваних процесів.

main.py python

```python
import sys
from functools import partial
from multiprocessing import Process
from timeit import timeit
```

```
REPEAT_NUMBER = int(sys.argv[1])
CHULD_PROCESSES_NUMBER = int(sys.argv[2])
REPEATS_PER_PROCESS = int(REPEAT_NUMBER / CHULD_PROCESSES_NUMBER)

def cpu_bound_function(repeat_num):
    for num in range(repeat_num):
        pow(num, 2)

def main():
    processes = []
    for num in range(CHULD_PROCESSES_NUMBER):
        p = Process(target=cpu_bound_function, args=(REPEATS_PER_PROCESS,))
        p.start()
        processes.append(p)
    for process in processes:
        process.join()

print(timeit(main, number=1))
```

Дана версія програми приймає два аргументи. Перший — кількість повторень CPU-bound операції, а другий — кількість потоків, на які розділяти виконання.

Для порівняння швидкості виконання спершу виконаємо версію програми з послідовним виконанням мільйону операцій.

terminal

```
$ python main.py 1000000
0.5813515731133521
```

Програма виконалась за близько півсекунди. Тепер використаємо версію програми з багатопроцесністю з різною кількістю дочірніх процесів.

terminal

```
$ python main.py 1000000 2
0.2858404382131994
$ python main.py 1000000 5
0.22751226602122188
$ python main.py 1000000 10
0.23588262032717466
$ python main.py 1000000 100
0.3526499909348786
```

```
$ python main.py 1000000 1000
1.8593445606529713
```

Цього разу дійсно видно пришвидшення роботи програми. Найефективнішою програма була при п’яти дочірніх процесах, при їх подальшому збільшенню загальна тривалість виконання почала збільшуватись, перевищивши тривалість послідовного виконання при використання тисячі процесів. Оптимальна кількість потоків для пришвидшення роботи конкретної програми залежить від самої програми та має визначатись експериментально.

Серіалізація даних для передачі між процесами

В прикладі вище при використанні класу “Process”, ми передавали аргументи для функції, через аргумент “args”. Окрім аргументу “args”, “Process.submit” приймай й аргумент “kwargs”. Аргументи “args” та “kwargs” приймають позиційні та іменовані аргументи відповідно.

Особливістю багатопроцесності є те, що батьківський та дочірний процеси не мають доступу до спільної пам’яті, тому будь-яка комунікація між ними, в тому числі і передача самої функції, що необхідно виконати та аргументів для неї відбувається через потоки стандартного вводу-виводу. Щоб передати функції та їх аргументи через потоки стандартного вводу-виводу, потрібно їх серіалізувати (закодувати) в батьківському процесі, та десеріалізувати (розкодувати) в дочірньому процесі. При використанні класу “Process” таким форматом серіалізації є pickle.

Pickle — це формат представлення Python-об’єктів у вигляді набору байтів [*]. Сереалізуючи Python дані, наприклад, у формат JSON, ми обмежеті тільки певним переліком типів. З pickle можна серіалізувати практично будь-які Python-об’єкти: числа, стрічки, списки, кортежі, словники, об’єкти класів (разом з методами), функції, класи та інші. В Python для роботи з форматом pickle використовується модуль зі стандартної бібліотеки “pickle”, що має інтерфейс схожий до інтерфейсу модуля “json”.

* pickle — Python object serialization — Python 3.8.0 documentation (https://docs.python.org/3/library/pickle.html)

Тим не менше не всі типи об’єктів можуть бути серіалізованими в pickle. В загальному випадку в pickle можуть бути серіалізованими типи, що мають визначений метод “__reduce__”. На практиці можливість чи неможливість серіалізації тих чи інших даних часто визначається на практиці.

Прикладом типу, що не можуть бути серіалізованим є екземпляри типу “file”. У випадках коли виникає необхідність передати у дочірній процес дані, що не можуть бути серіалізованими варто розглянути можливість передачі даних з яких відповідні об’єкти можуть бути створені, наприклад у випадку з файлом можна передати шлях до файлу і відкрити його вже в дочірньому процесі.

Повернення даних з дочірніх процесів

Як було видно з прикладів раніше функція, що виконувалась в окремому процесі, не повертала ніякого значення. Якщо таке значення і поверталося б, то воно було б втрачене і його неможливо було б дістати. Також змінювані аргументи, що були передані не будуть змінені в батьківському процесі при їх модифікації в дочірньому процесі.

Пакет “multiprocessing” має два способи для комунікації дочірнього процесу з батьківським. Перший — використання об’єктів спільної пам’яті (multiprocessing.Value та multiprocessing.Array). Другий — використання, так званих, менеджерів (multiprocessing.Manager)

Об’єкти спільної пам’яті досить обмежені порівняно з вбудованими типами в Python, вони мають статичний тип та працювати з ними не так зручно. Клас “Value” приймає аргументи типу та початкового значення. Тип вказується не з вбудованих типів Python а з типів з модуля “ctypes”, серед яких є такі типи як “c_int32”, “c_float”, “c_double”, “c_char” та інші. Клас “Array” приймає першим аргументом тип елементів масиву, а другим — послідовність (список, кортеж чи генератор), що буде визначати розмір масиву (що є фіксованим) та початкові значення. Приклад використання даних класів наведено нижче.

main.py python

```python
import ctypes
from multiprocessing import Process, Value, Array

def f(n, a):
    n.value = 3.1415927
    for i in range(len(a)):
        a[i] = -a[i]

if __name__ == '__main__':
    num = Value(ctypes.c_double, 0.0)
    arr = Array(ctypes.c_int32, range(10))

    p = Process(target=f, args=(num, arr))
    p.start()
    p.join()

    print(num.value)
    print(arr[:])
```

Запустимо дану програму:

terminal

```
$ python main.py
3.1415927
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

З виводу програми видно, що початкові значення були змінені в дочірньому процесі та нові значення доступні в батьківському процесі.

Про те як використовуються менеджери можна дізнатись з офіційної документації пакету “multiprocessing” [*].

* Sharing state between processes, Server process (https://docs.python.org/3/library/multiprocessing.html#sharing-state-between-processes)

Пул процесів, клас multiprocessing.Pool

Як було показано раніше, використання великої кількості процесів, що виконують відносно просту роботу, може бути не надто ефективним підходом. Більше того кожен новий процес додатково навантажує процесор та використовує оперативну пам’ять. В такому випадку для обмеження максимальної кількості дочірніх процесів використовують пул процесів. Пул процесів працює аналогічно пулу потоків. Пул процесів має розмір, який визначає максимальну кількість дочірніх процесів, що можуть одночасно виконуватись в пулі. При виконанні функцій в пулі, функції розподіляються по процесах. Якщо кількість таких функцій більша за розмір пулу, то функції, яким не вистачило процесів, чекають поки звільниться місце в пулі (коли якась з функцій, що виконується в пулі, завершиться).

Програмний інтерфейс “multiprocessing.Pool” аналогічний до “multiprocessing.pool.ThreadPool”. Нижче наведений приклад використання методу “map” класу “multiprocessing.Pool”.

main.py python

```python
import sys
from multiprocessing import Pool
from timeit import timeit

REPEAT_NUMBER = int(sys.argv[1])
POOL_SIZE = int(sys.argv[2])

def cpu_bound_function(num):
    pow(num, 2)

def main():
```

```python
    with Pool(POOL_SIZE) as pool:
        pool.map(cpu_bound_function, range(REPEAT_NUMBER))

print(timeit(main, number=1))
```

Виконаємо дану програму виконавши 10 мільйонів піднесень у степінь при різних розмірах пулу в якому ці операції будуть виконуватись.

terminal

```
$ python main.py 10000000 1
6.843864448834211
$ python main.py 10000000 2
4.175003189127892
$ python main.py 10000000 5
4.029894969891757
$ python main.py 10000000 10
3.8264315840788186
$ python main.py 10000000 20
3.665095399133861
$ python main.py 10000000 50
3.8192946300841874
$ python main.py 10000000 100
3.9792886837385595
$ python main.py 10000000 200
4.280887830071151
```

З виводу ачимо, що, як і очікувалось, при збільшенні розміру пулу, тривалість роботи програми зменшується, проте з певного моменту подальше збільшення розміру пулу має негативний ефект. Оптимальний розмір пулу залежить від конкретного випадку та має визначатись експериментально.

Одразу ж постає питання, чи пул перевикористовує процеси процеси в пулі чи створює нові. Перевіримо це наступною програмою.

main.py

```python
import os
from multiprocessing import Pool

def print_process_id(_unused_agument):
    pid = os.getpid()
    print(f'PID: {pid}')
```

```
if __name__ == '__main__':
    with Pool(1) as pool:
        pool.map(print_process_id, range(5))
```

Дана програма використовує пул розміром 1 в якому виконує задану функцію 5 разів. Задана функція виводить в термінал ідентифікатори дочірніх процесів. Якщо ідентифікатори будуть однаковими, це буде знаком, що процес перевикористовується. Виконаємо дану програму.

terminal

```
$ python main.py
PID: 64225
PID: 64225
PID: 64225
PID: 64225
PID: 64225
```

З виводу бачимо, що ідентифікатори процесів однакові, а отже процеси в пулі таки перевикористовуються.

Асинхронне виконання та пакет asyncio

terminal

```
$ python
Python 3.6.3 (default, Oct 14 2017, 15:56:35)
[GCC 4.2.1 Compatible Apple LLVM 9.0.0 (clang-900.0.37)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Python REPL

```
>>> print("Hello, world")
Hello, world
>>> def sum(a, b):
...     return a + b
...
```

```
>>> sum(2, 3)
5
```

main.py python

```
def intersection(a1, b1, c1, a2, b2, c2):
    ...
```