**Модуль 3: Багатопоточність і процеси в Python**

**Заняття 2: Процеси в Python**

**Пакет multiprocessing**

Пакет [multiprocessing](https://docs.python.org/3.8/library/multiprocessing.html) — це пакет для виконання коду в окремих процесах з інтерфейсом подібним до інтерфейсу пакета **threading.**

Основна причина появи **multiprocessing** — це GIL (Global Interpreter Lock) і той факт, що **threading** API не дозволяє розпаралелювати CPU-bound завдання. Оскільки в один момент часу завжди виконується код тільки в одному потоці, навіть на багатоядерних сучасних процесорах, отримати приріст продуктивності для завдань, пов'язаних з інтенсивними обчисленнями, за допомогою **threading** не вийде.

Щоб виконувати обчислення дійсно паралельно там, де це дозволяє обладнання, в Python використовуються окремі процеси. Так, у кожному окремому процесі буде запущено свій інтерпретатор Python зі своїм GIL.

Для використання процесів необхідно імпортувати клас Process модуля multiprocessing. З ним можна працювати декількома способами:

1. У процесі створення екземпляра класу Process іменованому аргументу target передати функцію, яка буде виконуватися в окремому процесі
2. Реалізувати похідний клас від класу Process та перевизначити метод run

Розглянемо приклад:

from multiprocessing import Process

import logging

from time import sleep

logger = logging.getLogger()

stream\_handler = logging.StreamHandler()

logger.addHandler(stream\_handler)

logger.setLevel(logging.DEBUG)

class MyProcess(Process):

def \_\_init\_\_(self, group=None, target=None, name=None, args=(), kwargs=None, \*, daemon=None):

super().\_\_init\_\_(group=group, target=target, name=name, daemon=daemon)

self.args = args

def run(self) -> None:

logger.debug(self.args)

def example\_work(params):

sleep(0.5)

logger.debug(params)

if \_\_name\_\_ == '\_\_main\_\_':

processes = []

for i in range(3):

pr = Process(target=example\_work, args=(f"Count process function - {i}", ))

pr.start()

processes.append(pr)

for i in range(2):

pr = MyProcess(args=(f"Count process class - {i}",))

pr.start()

processes.append(pr)

[el.join() for el in processes]

[print(el.exitcode, end=' ') for el in processes]

logger.debug('End program')

Виведення буде наступним:

('Count process class - 1',)

('Count process class - 0',)

Count process function - 0

Count process function - 1

Count process function - 2

End program

0 0 0 0 0

У цьому прикладі ми створили п'ять процесів, у трьох з яких виконали функцію example\_work, а у двох — це клас MyProcess, який наслідується від класу Process. У процесів є код завершення роботи (0 означає успішне завершення роботи у штатному режимі). І після завершення роботи атрибут exitcode містить код завершення. В іншому API multiprocessing багато в чому повторює threading.

https://docs.python.org/3.8/library/multiprocessing.html#programming-guidelines - [Загальні зауваження та поради](https://docs.python.org/3.8/library/multiprocessing.html#programming-guidelines) при роботі з процесами

**Contexts and start methods**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-03/module-03-02/intro#contexts-and-start-methods)Залежно від платформи **multiprocessing** підтримує 3 способи створення нового процесу:

* spawn — запускає новий процес Python, наслідуються лише ресурси, необхідні для запуску run(). Присутній в Unix і Windows. Є способом за замовчуванням для Windows і macOS.
* fork — дочірній процес, що є точною копією батьківського (включаючи всі потоки), доступний тільки на Unix. За замовчуванням використовується на Unix. Зробити безпечний fork досить проблематично і це може бути причиною неочевидних проблем.
* forkserver — створюється процес-фабрика (сервер для породження процесів за запитом). Наслідуються тільки необхідні ресурси, що використовуються fork для запуску нового процесу, але завдяки однопотоковій реалізації процесу-фабрики, це робиться безпечно. Доступний тільки на Unix платформах з підтримкою передачі файлових дескрипторів через pipes (що може суперечити безпековій політиці на багатьох системах).

Для вибору методу використовується multiprocessing.set\_start\_method(method)

import multiprocessing

...

if \_\_name\_\_ == '\_\_main\_\_':

multiprocessing.set\_start\_method('forkserver')

...

У цьому прикладі ми вибрали forkserver механізм створення дочірніх процесів.

**Interprocess communication**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-03/module-03-02/intro#interprocess-communication)Для міжпроцесорної взаємодії виокремлюють наступні інструменти:

* файли;
* сокети;
* канали (всі POSIX ОС);
* роздільна пам'ять (всі POSIX ОС);
* семафори (всі POSIX ОС);
* сигнали або переривання (крім Windows);
* семафори (всі POSIX ОС);
* черга повідомлень;

Найбільшу складність при роботі з процесами представляє обмін даними між процесами, оскільки у кожного процесу своя ізольована область пам'яті. Механізми обміну даними залежать від ОС (Операційної Системи). Найуніверсальніший — файли. Але ви також можете скористатися мережевими інтерфейсами (localhost), примітивами на основі мережевих інтерфейсів (pipe) та загальною пам'яттю, де це можливо.

У будь-якому разі, крім загальної пам'яті, для обміну даними між процесами всі об'єкти серіалізуються та десеріалізуються. Цей додатковий крок створює навантаження на CPU.

Найшвидшим та найекономнішим з погляду ресурсів способом обміну даними є спільна пам'ять.

**Спільна пам'ять**

Спільна пам'ять підтримується не всіма операційними системами та може бути заборонена політикою безпеки.

Щоб створити область спільної пам'яті, потрібно вказати ОС, скільки пам'яті потрібно виділити. Для обчислення обсягу пам'яті обов'язково потрібно вказати тип даних, який буде використовуватись, та кількість елементів для складних типів. Крім того, механізми обмеження доступу до спільного ресурсу також потрібно забезпечити самостійно, інакше дані можна зіпсувати при спробі одночасного доступу для зміни із різних процесів.

from multiprocessing import Process, Value, RLock, current\_process

from time import sleep

import logging

import sys

logger = logging.getLogger()

stream\_handler = logging.StreamHandler()

logger.addHandler(stream\_handler)

logger.setLevel(logging.DEBUG)

def worker(val: Value):

logger.debug(f'Started {current\_process().name}')

sleep(1)

with val.get\_lock():

val.value += 1

logger.debug(f'Done {current\_process().name}')

sys.exit(0)

if \_\_name\_\_ == '\_\_main\_\_':

lock = RLock()

value = Value('d', 0, lock=lock)

pr1 = Process(target=worker, args=(value, ))

pr1.start()

pr2 = Process(target=worker, args=(value, ))

pr2.start()

pr1.join()

pr2.join()

print(value.value) # 2.0

Виведення:

Started Process-1

Started Process-2

Done Process-1

Done Process-2

2.0

У цьому прикладі ми скористалися механізмом спільної пам'яті Value. Тип даних було обрано десятковий ('d'). Докладніше про доступні типи та їх назви можна дізнатися з **документацією:**

 https://docs.python.org/3.8/library/ctypes.html#fundamental-data-types**.**

Щоб створити спільну пам'ять для процесів, необхідно визначити тип даних, який буде знаходитися у виділеній області пам'яті. Це порушує звичний для Python підхід, коли не потрібно думати про те, який тип даних буде використовуватися і скільки місця він може займати.

Давайте розберемо складніший приклад, з використанням структур у спільній пам'яті:

from multiprocessing import Process, RLock, current\_process

from multiprocessing.sharedctypes import Value, Array

from ctypes import Structure, c\_double

import logging

logger = logging.getLogger()

stream\_handler = logging.StreamHandler()

logger.addHandler(stream\_handler)

logger.setLevel(logging.DEBUG)

class Point(Structure):

\_fields\_ = [('x', c\_double), ('y', c\_double)]

def modify(num: Value, string: Array, arr: Array):

logger.debug(f'Started {current\_process().name}')

logger.debug(f"Change num: {num.value}")

with num.get\_lock():

num.value \*\*= 2

logger.debug(f"to num: {num.value}")

with string.get\_lock():

string.value = string.value.upper()

with arr.get\_lock():

for a in arr:

a.x \*\*= 2

a.y \*\*= 2

logger.debug(f'Done {current\_process().name}')

if \_\_name\_\_ == '\_\_main\_\_':

lock = RLock()

number = Value(c\_double, 1.5, lock=lock)

string = Array('c', b'hello world', lock=lock)

array = Array(Point, [(1, -6), (-5, 2), (2, 9)], lock=lock)

p = Process(target=modify, args=(number, string, array))

p2 = Process(target=modify, args=(number, string, array))

p.start()

p2.start()

p.join()

p2.join()

print(number.value)

print(string.value)

print([(arr.x, arr.y) for arr in array])

Виведення буде:

Started Process-2

Change num: 1.5

to num: 2.25

Done Process-2

Started Process-1

Change num: 2.25

to num: 5.0625

Done Process-1

5.0625

b'HELLO WORLD'

[(1.0, 1296.0), (625.0, 16.0), (16.0, 6561.0)]

У цьому прикладі ми створили:

* структуру Point, яка описує координати точки на площині;
* дробове число number;
* рядкову змінну string (підтримуються тільки byte-рядки);
* масив array, який містить координати точок відповідно до структури Point.

Зверніть увагу, для опису полів структури їх потрібно помістити в список кортежів [\_fields\_](https://docs.python.org/3/library/ctypes.html#structures-and-unions), де кожен кортеж — це ім'я та тип поля.

Масив Array поводиться багато в чому як список і дозволяє зберігати в ньому різнотипні дані, але його розмір статичний і додавати/видаляти елементи не можна. Так само як і змінювати тип існуючих.

Також у структурі даних ми передали механізм блокування через параметр lock. Як Value, так і Array забезпечують блокування ресурсу, до якого можна отримати доступ, щоб як читати, так і оновлювати дані.

**Менеджер ресурсів**

Вимогливіший до ресурсів, але й зручніший у використанні механізм обміну даними між процесами — це[**Менеджер ресурсів**](https://docs.python.org/3/library/multiprocessing.html#managers) (https://docs.python.org/3/library/multiprocessing.html#managers). Основна перевага — можливість працювати по всій мережі та реалізувати розподілені обчислення між кількома комп'ютерами в одній мережі, реалізація Python-like списків та словників.

Недоліки:

* Необхідність синхронізувати доступ до загальних ресурсів;
* Обмеження типів, що підтримуються;
* Складне API.

Розглянемо наступний приклад:

from multiprocessing import Process, Manager, current\_process

from random import randint

from time import sleep

import logging

logger = logging.getLogger()

stream\_handler = logging.StreamHandler()

logger.addHandler(stream\_handler)

logger.setLevel(logging.DEBUG)

def worker(delay, val: Manager):

name = current\_process().name

logger.debug(f'Started: {name}')

sleep(delay)

val[name] = current\_process().pid

logger.debug(f'Done: {name}')

if \_\_name\_\_ == '\_\_main\_\_':

with Manager() as manager:

m = manager.dict()

processes = []

for i in range(5):

pr = Process(target=worker, args=(randint(1, 3), m))

pr.start()

processes.append(pr)

[pr.join() for pr in processes]

print(m)

Виведення:

Started: Process-2

Started: Process-3

Started: Process-5

Started: Process-4

Started: Process-6

Done: Process-3

Done: Process-5

Done: Process-2

Done: Process-6

Done: Process-4

{'Process-3': 7444, 'Process-5': 15976, 'Process-2': 15564, 'Process-6': 18896, 'Process-4': 14244}

У цьому прикладі ми запустили п'ять процесів і додали до словника m, для кожного процесу його pid — ідентифікатор процесу. Все це було створено та управлялося менеджером Manager.

Але є важливе зауваження. Проксі-об'єкти Manager не можуть поширювати зміни, внесені до об'єктів, що змінюються всередині контейнера. Іншими словами, якщо у вас є об'єкт manager.list(), будь-які зміни в самому керованому списку розповсюджуються на всі інші процеси. Але якщо у вас є звичайний список Python всередині цього списку, будь-які зміни у внутрішньому списку не поширюються, тому що менеджер не має можливості виявити зміни.

Щоб розповсюдити зміни, ви також повинні використовувати об'єкти manager.list() для вкладених списків. (необхідний Python 3.6 або вище) або вам потрібно безпосередньо змінити об'єкт manager.list() ([див. примітку](https://docs.python.org/3.5/library/multiprocessing.html#multiprocessing.managers.SyncManager.list)) .

**Pipe (канал)**

Механізм каналів реалізований поверх сокетів (мережеві інтерфейси) і дає зручніший інтерфейс. Канали доступні на всіх POSIX системах (Linux, Mac OS, FreeBSD, Unix).

Канал — пара об'єктів — кінці каналу. Ви можете асинхронно писати в канал (надсилати повідомлення) та читати з каналу (отримувати повідомлення на іншому кінці). Повідомлень у каналі може бути більше одного одночасно і порядок повідомлень гарантується. Канали реалізують обмін між двома процесами, не можна використовувати один і той самий кінець каналу в кількох процесах.

Клас [Pipe](https://docs.python.org/3/library/multiprocessing.html#multiprocessing.Pipe) модуля multiprocessing повертає парний кортеж (conn1, conn2), що складається з об'єктів Connection, що представляють кінці одного каналу.

Клас Pipe приймає аргумент дуплекс duplex, який за замовчуванням дорівнює True. Тоді канал є двоспрямованим, якщо ж duplex=False, тоді канал є односпрямованим і тоді conn1 використовується для приймання повідомлень, а conn2 для надсилання повідомлень.

Об'єкти conn1 та conn2 мають ряд методів, основні це:

* send — відправляє об'єкт на інший кінець з'єднання
* recv — повертає об'єкт, відправлений з іншого кінця з'єднання
* close закриває з'єднання

Давайте розглянемо приклад реалізації каналу в Python за допомогою класу Pipe.

from multiprocessing import Pipe, Process, current\_process

from time import sleep

import sys

import logging

logger = logging.getLogger()

stream\_handler = logging.StreamHandler()

logger.addHandler(stream\_handler)

logger.setLevel(logging.DEBUG)

recipient1, sender1 = Pipe()

recipient2, sender2 = Pipe()

def worker(pipe: Pipe):

name = current\_process().name

logger.debug(f'{name} started...')

val = pipe.recv()

logger.debug(val\*\*2)

sys.exit(0)

if \_\_name\_\_ == '\_\_main\_\_':

w1 = Process(target=worker, args=(recipient1, ))

w2 = Process(target=worker, args=(recipient2, ))

w1.start()

w2.start()

sender1.send(8)

sleep(1)

sender2.send(16)

У цьому прикладі ми пишемо в канали з батьківського процесу sender1.send(8) та sender2.send(16), а читаємо з дочірніх pipe.recv() всередині функції worker.

Виведення:

Process-1 started...

64

Process-2 started...

256

У канал можна відправити будь-який тип даних, який можна перетворити на byte-рядок за допомогою pickle. Не можна надсилати функції або інші об'єкти, які не серіалізуються.

**Черги завдань**

Окремий механізм обміну даними між процесами — це черги. Черга дозволяє "покласти" дані для обробки одним із "працівників" і потім проконтролювати, коли "працівник" завершив обробку. Черги — це засіб комунікації між одним відправником, найчастіше його називають master, та будь-якою кількістю одержувачів повідомлень, часто їх називають slave або у нейтральнішому тоні — worker.

Об'єктів у черзі може бути більше одного. Черга може бути обмежена за розміром, якщо це потрібно. Черга гарантує порядок повідомлень та неможливість отримання одного повідомлення кількома одержувачами.

Черги в Python реалізовані у класах Queue та JoinableQueue:

from multiprocessing import Queue, Process, current\_process

from time import sleep

import sys

import logging

logger = logging.getLogger()

stream\_handler = logging.StreamHandler()

logger.addHandler(stream\_handler)

logger.setLevel(logging.DEBUG)

q = Queue()

def worker(queue: Queue):

name = current\_process().name

logger.debug(f'{name} started...')

val = queue.get()

logger.debug(f'{name} {val\*\*2}')

sys.exit(0)

if \_\_name\_\_ == '\_\_main\_\_':

w1 = Process(target=worker, args=(q, ))

w2 = Process(target=worker, args=(q, ))

w1.start()

w2.start()

q.put(8)

sleep(1)

q.put(16)

Виведення:

Process-1 started...

Process-1 64

Process-2 started...

Process-2 256

У цьому прикладі ми створили два процеси w1 та w2, які беруть із черги q собі завдання і виконують їх. Наш основний процес, що виконує роль master, кладе завдання у чергу q, але ніяк не контролює їх виконання, а процеси worker читають повідомлення з черги та завершуються.

Для контролю процесу завершення завдань можна використовувати клас JoinableQueue:

from multiprocessing import JoinableQueue, Process, current\_process

from time import sleep

import sys

import logging

logger = logging.getLogger()

stream\_handler = logging.StreamHandler()

logger.addHandler(stream\_handler)

logger.setLevel(logging.DEBUG)

jq = JoinableQueue()

def worker(jqueue: JoinableQueue):

name = current\_process().name

logger.debug(f'{name} started...')

val = jqueue.get()

logger.debug(f'{name} {val\*\*2}')

sleep(1)

jqueue.task\_done()

sys.exit(0)

if \_\_name\_\_ == '\_\_main\_\_':

w1 = Process(target=worker, args=(jq, ))

w2 = Process(target=worker, args=(jq, ))

w1.start()

w2.start()

jq.put(8)

sleep(1)

jq.put(16)

jq.join()

print('Finished')

Виведення:

Process-2 started...

Process-2 64

Process-1 started...

Process-1 256

Finished

У цьому прикладі master-процес (основний процес) очікує виконання завдань від робочих процесів за допомогою виклику методу jq.join() у екземпляра черги JoinableQueue. Самі worker-процеси повідомляють master про успішне виконання завдання через виклик методу черги jqueue.task\_done(). Це дозволяє тримати в курсі master-процес про стан завершення робочого процесу та рядок коду print('Finished') виконається тільки тоді, коли всі робочі процеси будуть завершені.

! TIP

Процеси використовують для синхронізації також весь набір класів, який ми розглянули для потоків, — це: RLock, Event, Condition, Semaphore та Barrier. Принцип їх застосування той самий, але імпортуються класи з пакету multiprocessing. Ми не розглядали приклади їх застосування, оскільки, як ми вже сказали, принцип роботи у них такий самий, як і у потоках, і нічим не відрізняється.

**Пул процесів**

**Створення процесів за допомогою пакета multiprocessing**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-03/module-03-02/pool#%D1%81%D1%82%D0%B2%D0%BE%D1%80%D0%B5%D0%BD%D0%BD%D1%8F-%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%96%D0%B2-%D0%B7%D0%B0-%D0%B4%D0%BE%D0%BF%D0%BE%D0%BC%D0%BE%D0%B3%D0%BE%D1%8E-%D0%BF%D0%B0%D0%BA%D0%B5%D1%82%D0%B0-multiprocessing)Для спрощення комунікації у пакеті multiprocessing є клас, який реалізує пул процесів за аналогією з concurrent.futures.

Основне застосування — це виконання паралельно однакових завдань із деяким набором однотипних вхідних даних.

from multiprocessing import Pool, current\_process

import logging

logger = logging.getLogger()

stream\_handler = logging.StreamHandler()

logger.addHandler(stream\_handler)

logger.setLevel(logging.DEBUG)

def worker(x):

logger.debug(f"pid={current\_process().pid}, x={x}")

return x\*x

if \_\_name\_\_ == '\_\_main\_\_':

with Pool(processes=2) as pool:

logger.debug(pool.map(worker, range(10)))

Результатом виконання цього коду буде приблизно наступне:

pid=14592, x=0

pid=14592, x=1

pid=14592, x=2

pid=14592, x=3

pid=14592, x=4

pid=14592, x=5

pid=14592, x=6

pid=14592, x=7

pid=14592, x=8

pid=14592, x=9

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

Пул процесів з multiprocessing дає більше контролю, ніж пул з concurrent.futures.

Основні можливості:

* розбиває вхідну послідовність на блоки та виконує паралельну обробку поблоково, так можна зменшити обсяг використовуваної пам'яті;
* асинхронне виконання трохи прискорює отримання результатів, якщо порядок не важливий;
* передача кортежу аргументів у target-функцію;

Детально з можливостями можна ознайомитись на [сторінці](https://docs.python.org/3.8/library/multiprocessing.html#module-multiprocessing.pool) офіційної документації https://docs.python.org/3.8/library/multiprocessing.html#module-multiprocessing.pool.

**Створення процесів за допомогою пакета concurrent**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-03/module-03-02/pool#%D1%81%D1%82%D0%B2%D0%BE%D1%80%D0%B5%D0%BD%D0%BD%D1%8F-%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%96%D0%B2-%D0%B7%D0%B0-%D0%B4%D0%BE%D0%BF%D0%BE%D0%BC%D0%BE%D0%B3%D0%BE%D1%8E-%D0%BF%D0%B0%D0%BA%D0%B5%D1%82%D0%B0-concurrent)Пакет concurrent.futures також реалізує API Executor для пулу процесів у класі [ProcessPoolExecutor](https://docs.python.org/3/library/concurrent.futures.html#processpoolexecutor).

Основні можливості обмежені API Executor. Зручно використовувати ProcessPoolExecutor там, де потрібно виконати CPU-bound завдання в async коді та реалізовано саме для підтримки виконання блокуючих CPU-bound завдань в async застосунках (з async ми познайомимося в "Модуль 5: Асинхронне програмування в Python").

Зараз поки що розглянемо приклад виконання CPU-bound завдання:

import concurrent.futures

import math

PRIMES = [

112272535095293,

112582705942171,

112272535095293,

115280095190773,

115797848077099,

1099726899285419]

def is\_prime(n):

if n < 2:

return False

if n == 2:

return True

if n % 2 == 0:

return False

sqrt\_n = int(math.floor(math.sqrt(n)))

for i in range(3, sqrt\_n + 1, 2):

if n % i == 0:

return False

return True

if \_\_name\_\_ == '\_\_main\_\_':

with concurrent.futures.ProcessPoolExecutor(4) as executor:

for number, prime in zip(PRIMES, executor.map(is\_prime, PRIMES)):

print('%d is prime: %s' % (number, prime))

Наприклад, цей код для пошуку простих чисел виконується у чотирьох процесах і поверне:

112272535095293 is prime: True

112582705942171 is prime: True

112272535095293 is prime: True

115280095190773 is prime: True

115797848077099 is prime: True

1099726899285419 is prime: False