# Багатопоточність: Теорія та Концепції

## 1.1. Паралелізм vs. Конкурентність

Перш ніж зануритися в багатопоточність, важливо розуміти різницю між **паралелізмом** та **конкурентністю**.

- **Конкурентність (Concurrency):** Це здатність системи обробляти кілька завдань "одночасно", але не обов'язково в один і той же момент. Завдання можуть перемикатися між собою (наприклад, за допомогою перемикання контексту), створюючи ілюзію одночасності. Це як жонглювання кількома м'ячами одним жонглером.
    - *Приклад:* Ваш комп'ютер може виконувати багато програм одночасно (браузер, музичний плеєр, текстовий редактор), перемикаючись між ними дуже швидко.
- **Паралелізм (Parallelism):** Це фактичне одночасне виконання кількох завдань у той самий момент часу, зазвичай на різних процесорних ядрах або окремих процесорах. Це як кілька жонглерів, кожен з яких жонглює своїми м'ячами.
    - *Приклад:* Програма, яка використовує всі 4 ядра вашого процесора для виконання 4 частин одного обчислення одночасно.

**Багатопоточність у Python (через GIL) зазвичай забезпечує конкурентність, а не справжній паралелізм для CPU-bound завдань.**

## 1.2. Процеси (Processes)

**Процес** — це незалежна одиниця виконання програми. Кожен процес має свій власний:

- **Адресний простір:** Окрема область пам'яті, ізольована від інших процесів.
- **Ресурси:** Власний набір файлових дескрипторів, мережевих з'єднань тощо.
- **Контекст виконання:** Включає стан процесора, регістри, стек.

Процеси ізольовані один від одного, що робить їх дуже стабільними: збій в одному процесі зазвичай не впливає на інші. Спілкування між процесами (IPC - Inter-Process Communication) вимагає спеціальних механізмів (черги, пайпи, спільна пам'ять).

## 1.3. Потоки (Threads)

**Потік (thread)** — це легковаговий потік виконання всередині процесу. Кілька потоків можуть існувати в одному процесі і **ділити один і той самий адресний простір пам'яті** та ресурси процесу.

- **Спільна пам'ять:** Потоки мають доступ до одних і тих же глобальних змінних та об'єктів у пам'яті. Це робить обмін даними між потоками дуже швидким, але й створює потенційні проблеми (див. нижче).
- **Легковаговість:** Створення та перемикання між потоками зазвичай швидше та вимагає менше ресурсів, ніж створення та перемикання між процесами.
- **Контекст:** Кожен потік має свій власний стек викликів, лічильник команд та регістри.

**Клас threading.Thread**

Модуль threading є вбудованим модулем у Python, який надає можливість створювати та керувати потоками в програмі. Потоки представляють собою окремі шляхи виконання, які можуть працювати паралельно, дозволяючи виконувати кілька завдань одночасно. Клас Thread дозволяє створювати нові потоки. Він приймає функцію, яку ви хочете виконати у новому потоці.

**Приклад:**

```
import threading

def print_numbers():
    for i in range(5):
        print(i)

thread = threading.Thread(target=print_numbers)
thread.start()  # Запускає потік
# thread.join() # У реальному коді тут може знадобитися join() для
очікування завершення потоку
```

## 1.4. Навіщо багатопоточність? (Випадки використання у веб-розробці)

Багатопоточність особливо корисна для **I/O-bound (пов'язаних з вводом/виводом)** завдань. Це завдання, які більшу частину часу чекають на завершення операцій вводу/виводу, а не на обчислення процесора.

- **Веб-розробка:**
    - **Запити до зовнішніх API:** Коли ваш веб-сервер робить запит до іншого сервісу (наприклад, платіжного шлюзу, стороннього API), він чекає на відповідь. Потік може перейти в "сплячий" режим, дозволяючи іншому потоку обробляти інший запит користувача.

    - **Доступ до бази даних:** Читання або запис даних у базу даних.
    - **Завантаження/вивантаження файлів:** Операції з файловою системою.
    - **Мережеві операції:** Очікування на відповідь від клієнта або іншого сервера.
    - **Паралельна обробка запитів:** Веб-сервери часто використовують потоки (або асинхронність) для одночасного обслуговування кількох клієнтських запитів.
- **Приклад:** Уявіть веб-сервер, який отримує 100 запитів. Якщо кожен запит включає 1 секунду очікування на базу даних, то послідовна обробка займе 100 секунд. Якщо використовувати 10 потоків, які паралельно чекають на базу даних, загальний час може бути значно меншим.

## 1.5. Global Interpreter Lock (GIL)

**Global Interpreter Lock (GIL)** — це механізм у CPython (стандартна реалізація Python), який гарантує, що **лише один потік може виконувати байт-код Python одночасно**.

**Наслідки GIL:**

- **CPU-bound завдання:** Для завдань, які інтенсивно використовують процесор (наприклад, складні математичні обчислення), багатопоточність у Python **не забезпечує справжнього паралелізму**. Потоки все одно змагаються за один і той самий GIL, і фактично виконуються послідовно (хоч і швидко перемикаються).
- **I/O-bound завдання:** Для I/O-bound завдань GIL звільняється під час операцій вводу/виводу (наприклад, читання з файлу, мережевий запит). Це дозволяє іншим потокам виконуватися, поки один потік чекає на I/O. **Саме тому багатопоточність ефективна для веб-розробки в Python.**

**Обхід GIL:**

- **Багатопроцесність (multiprocessing):** Кожен процес має свій власний інтерпретатор Python та свій GIL, що дозволяє справжній паралелізм на кількох ядрах.
- **Використання C-розширень:** Бібліотеки, написані на C (наприклад, NumPy), можуть виконувати обчислення поза GIL.
- **Асинхронне програмування (asyncio):** Для I/O-bound завдань, де немає блокування потоку.

## 1.6. Проблеми синхронізації потоків

Оскільки потоки ділять спільну пам'ять, виникають проблеми, якщо кілька потоків

намагаються одночасно змінити одні й ті ж дані.

- **Стан гонки (Race Condition):** Виникає, коли кілька потоків намагаються отримати доступ до спільного ресурсу (змінної, об'єкта) і змінити його одночасно, і кінцевий результат залежить від того, в якому порядку ці потоки виконуються.
    - *Приклад:* Два потоки одночасно намагаються збільшити лічильник на 1. Замість збільшення на 2, він може збільшитись лише на 1, якщо операції читання/запису перекриваються.
- **Взаємне блокування (Deadlock):** Ситуація, коли два або більше потоків заблоковані назавжди, чекаючи на ресурси, які утримуються іншими потоками, що також заблоковані.
    - *Приклад:* Потік А має ресурс X і чекає на ресурс Y. Потік Б має ресурс Y і чекає на ресурс X. Жоден з них не може продовжити.

## 1.7. Примітиви синхронізації потоків

Для запобігання станам гонки та взаємним блокуванням використовуються примітиви синхронізації, які надає модуль threading.

- **Клас threading.Lock:**
    - Клас Lock використовується для синхронізації доступу до спільних ресурсів з декількох потоків. Дозволяє лише одному потоку одночасно отримати доступ до захищеної секції коду (критичної секції).
    - lock.acquire(): Блокує доступ. Якщо лок вже захоплений, потік чекає.
    - lock.release(): Звільняє лок.
    - Часто використовується з with lock:, що гарантує автоматичне звільнення локу.

    **Приклад:**

```
import threading

shared_variable = 0
lock = threading.Lock()

def increment():
    global shared_variable
    with lock: # Захоплює лок, виконує код, потім звільняє
        shared_variable += 1

threads = []
```

```
for _ in range(5):
    thread = threading.Thread(target=increment)
    threads.append(thread)
    thread.start()

for thread in threads:
    thread.join()

print("Загальна змінна:", shared_variable) # Буде 5
```

- **Клас threading.Event:**
  - Клас Event дозволяє потокам взаємодіяти за допомогою сигналів. Один потік сигналізує іншому, що певна подія сталася.
  - event.set(): Встановлює внутрішній прапорець у True, сигналізуючи іншим потокам.
  - event.clear(): Скидає прапорець у False.
  - event.wait(): Блокує потік, доки прапорець не буде встановлено у True.

**Приклад:**

```
import threading
import time

event = threading.Event()

def wait_for_event():
    print("Чекаю на подію...")
    event.wait() # Потік блокується тут
    print("Подія сталася!")

def set_event():
    time.sleep(2) # Імітуємо роботу
    print("Подія встановлена.")
    event.set() # Сигналізуємо

thread1 = threading.Thread(target=wait_for_event)
thread2 = threading.Thread(target=set_event)

thread1.start()
```

```
thread2.start()

thread1.join()
thread2.join()
```

- **Клас threading.Semaphore:**
  - Клас Semaphore дозволяє обмежити кількість потоків, які можуть одночасно виконувати певний блок коду.
  - Ініціалізується з лічильником (value). acquire() зменшує лічильник, release() збільшує.
  - Якщо лічильник дорівнює нулю, acquire() блокується.
  - Корисно для обмеження кількості одночасних підключень до бази даних або зовнішніх сервісів.

**Приклад:**

```
import threading
import time
import random

semaphore = threading.Semaphore(value=2) # Дозволяє 2 потоки одночасно

def limited_function(thread_id):
    print(f"Потік {thread_id}: Намагається отримати доступ...")
    with semaphore: # Захоплює семафор
        print(f"Потік {thread_id}: Початок виконання в критичній секції.")
        time.sleep(random.uniform(0.5, 1.5)) # Імітація роботи
        print(f"Потік {thread_id}: Кінець виконання в критичній секції.")

threads = []
for i in range(4): # 4 потоки, але лише 2 можуть працювати одночасно
    thread = threading.Thread(target=limited_function, args=(i+1,))
    threads.append(thread)
    thread.start()

for thread in threads:
    thread.join()
```

## 1.8. Коли використовувати багатопоточність vs. багатопроцесність?

- **Багатопоточність (threading):**
  - **I/O-bound завдання:** Очікування мережевих запитів, файлових операцій, запитів до БД.
  - **Простота обміну даними:** Потоки ділять пам'ять, що спрощує обмін даними (але вимагає синхронізації).
  - **Легковаговість:** Менші накладні витрати на створення та перемикання.
  - **Обмеження GIL:** Не для CPU-bound завдань у CPython.
- **Багатопроцесність (multiprocessing):**
  - **CPU-bound завдання:** Для використання всіх ядер процесора та досягнення справжнього паралелізму.
  - **Ізоляція:** Кожен процес є незалежним, що підвищує стабільність (збій одного не впливає на інші).
  - **Складніший обмін даними:** Потрібні спеціальні механізми IPC.
  - **Більші накладні витрати:** Створення процесів дорожче.

**Для веб-розробників у Python, багатопоточність часто є хорошим вибором для обробки паралельних I/O-bound операцій, тоді як для дуже інтенсивних обчислень краще підійде багатопроцесність.**

# Практичні Приклади Багатопоточності

```
import threading
import time
import datetime
import random
```

## Приклад 1: Базове створення та виконання потоків (threading.Thread)

Створення потоків за допомогою `threading.Thread` та запуск їх.

```
def task(name, duration):
    """Проста функція, що імітує виконання завдання."""
    print(f"[{datetime.datetime.now().strftime('%H:%M:%S')}] Потік '{name}': Починаю завдання на {duration} сек.")
    time.sleep(duration) # Імітація роботи
    print(f"[{datetime.datetime.now().strftime('%H:%M:%S')}] Потік '{name}': Завершую завдання.")

# Створення потоків
thread1 = threading.Thread(target=task, args=("Потік A", 2))
thread2 = threading.Thread(target=task, args=("Потік B", 1))

# Запуск потоків
thread1.start()
thread2.start()

# `join()` блокує головний потік, доки ці потоки не завершаться
thread1.join()
thread2.join()
```

## Завдання до Прикладу 1: Базове створення потоків

Створіть два потоки. Один потік має вивести "Привіт!" 3 рази з інтервалом 0.5 секунди,  інший - "Бувай!" 2 рази з інтервалом 0.8 секунди.

Відповідь

## Приклад 2: Передача аргументів потокам

Демонструє, як передавати аргументи до функцій, що виконуються в потоках.

```
def process_item(item_id, user_name, delay):
    """Імітує обробку елемента для користувача."""
    print(f"[{datetime.datetime.now().strftime('%H:%M:%S')}] Обробка
Item {item_id} для {user_name}...")
```

```
    time.sleep(delay)
    print(f"[{datetime.datetime.now().strftime('%H:%M:%S')}] Item
{item_id} для {user_name} оброблено.")

items_to_process = [
    (101, "Олег", 1.5),
    (102, "Марія", 0.8),
    (103, "Іван", 2.0)
]
threads = []
for item_id, user_name, delay in items_to_process:
    thread = threading.Thread(target=process_item, args=(item_id,
user_name, delay))
    threads.append(thread)
    thread.start()

for thread in threads:
    thread.join()
```

## Завдання до Прикладу 2: Передача аргументів потокам

Створіть функцію `calculate_square(number)`. Запустіть 3 потоки, кожен з яких обчислює квадрат різних чисел (наприклад, 5, 8, 12). Виведіть результат обчислення у функції потоку.

Відповідь

## Приклад 3: Багатопоточність для I/O-bound завдань (імітація веб-запитів)

Демонструє, як потоки можуть прискорити виконання завдань, що чекають на I/O.

```
def fetch_url_data(url):
    """Імітує завантаження даних з URL."""
    print(f"[{datetime.datetime.now().strftime('%H:%M:%S')}]
Завантажую дані з: {url}...")
    # У реальному житті тут був би requests.get(url)
    time.sleep(random.uniform(0.5, 2.0)) # Імітація затримки мережі
    data = f"Дані з {url}"
    print(f"[{datetime.datetime.now().strftime('%H:%M:%S')}]
Завантажено: {url}")
    return data

urls = [
    "http://example.com/page1",
    "http://example.com/page2",
    "http://example.com/page3",
    "http://example.com/page4",
]

start_time_io = time.perf_counter()
threads_io = []
for url in urls:
```

```
    thread = threading.Thread(target=fetch_url_data, args=(url,))
    threads_io.append(thread)
    thread.start()

for thread in threads_io:
    thread.join()
end_time_io = time.perf_counter()
print(f"Загальний час виконання (з потоками): {end_time_io -
start_time_io:.4f} секунд.")

# Для порівняння, як би це було послідовно:
print("\n--- Порівняння: Послідовне виконання ---")
start_time_seq = time.perf_counter()
for url in urls:
    fetch_url_data(url)
end_time_seq = time.perf_counter()
print(f"Загальний час виконання (послідовно): {end_time_seq -
start_time_seq:.4f} секунд.")
```

## Завдання до Прикладу 3: Багатопоточність для I/O-bound завдань

Імітуйте "завантаження" 5 файлів з різними випадковими затримками (від 0.1 до 1.0 секунди) за допомогою потоків. Виведіть повідомлення про початок та завершення завантаження кожного файлу.

Відповідь

## Приклад 4: Використання `threading.Lock` для запобігання станам гонки

Демонструє, як блокування може захистити спільний ресурс.

```
shared_counter = 0
counter_lock = threading.Lock() # Створюємо об'єкт блокування

def increment_counter_unsafe():
    """Небезпечна функція, що інкрементує лічильник без блокування."""
    global shared_counter
    current_value = shared_counter
    time.sleep(0.001) # Імітація затримки, щоб збільшити ймовірність
гонки
    shared_counter = current_value + 1

def increment_counter_safe():
    """Безпечна функція, що інкрементує лічильник з блокуванням."""
    global shared_counter
    with counter_lock: # Захоплюємо лок перед доступом до спільного
ресурсу
        current_value = shared_counter
```

```
        time.sleep(0.001)
        shared_counter = current_value + 1

num_increments = 1000

# Небезпечний сценарій
shared_counter = 0
threads_unsafe = []
for _ in range(num_increments):
    thread = threading.Thread(target=increment_counter_unsafe)
    threads_unsafe.append(thread)
    thread.start()

for thread in threads_unsafe:
    thread.join()
print(f"Небезпечний сценарій: Очікувано {num_increments}, Отримано
{shared_counter}") # Буде менше, ніж очікувано

# Безпечний сценарій
shared_counter = 0
threads_safe = []
for _ in range(num_increments):
    thread = threading.Thread(target=increment_counter_safe)
    threads_safe.append(thread)
    thread.start()

for thread in threads_safe:
    thread.join()
print(f"Безпечний сценарій: Очікувано {num_increments}, Отримано
{shared_counter}") # Буде точно {num_increments}
print("-" * 30)
```

## Завдання до Прикладу 4: Використання

## `threading.Lock`

Уявіть, що у вас є спільний список `shared_list`, до якого кілька потоків додають елементи. Виправте стан гонки, щоб усі елементи були додані коректно.

Відповідь

## Приклад 5: Використання `threading.Semaphore` для обмеження ресурсів

Обмеження кількості одночасних "з'єднань" до імітованої бази даних.

```
max_db_connections = 3
db_semaphore = threading.Semaphore(max_db_connections) # Дозволяє 3
одночасні доступи

def access_database(user_id):
    """Імітує доступ до бази даних."""
    print(f"[{datetime.datetime.now().strftime('%H:%M:%S')}]
Користувач {user_id}: Намагається отримати доступ до БД...")
    with db_semaphore: # Захоплюємо семафор
        print(f"[{datetime.datetime.now().strftime('%H:%M:%S')}]
Користувач {user_id}: Отримав доступ до БД. (Активних:
{max_db_connections - db_semaphore._value})")
        time.sleep(random.uniform(0.5, 2.0)) # Імітація роботи з БД
        print(f"[{datetime.datetime.now().strftime('%H:%M:%S')}]
Користувач {user_id}: Звільняє доступ до БД.")

print("\n--- Приклад 5: Обмеження ресурсів з Semaphore ---")
num_users = 10
threads_db = []
for i in range(num_users):
    thread = threading.Thread(target=access_database, args=(i,))
    threads_db.append(thread)
    thread.start()

for thread in threads_db:
    thread.join()
print("Всі користувачі завершили доступ до БД.")
```

## Завдання до Прикладу 5: Використання `threading.Semaphore`

Уявіть, що у вас є 5 "принтерів", і лише 3 з них можуть друкувати одночасно. Створіть семафор для обмеження доступу до принтерів. Імітуйте 10 завдань на друк.

Відповідь

## Приклад 6: Використання `threading.Event` для синхронізації

Один потік чекає, поки інший потік подасть сигнал.

```
start_event = threading.Event() # Створюємо об'єкт події

def worker_thread_event(name):
    print(f"Потік '{name}': Чекаю на сигнал початку...")
    start_event.wait() # Блокується, доки подія не буде встановлена
    print(f"Потік '{name}': Отримав сигнал. Починаю роботу!")
    time.sleep(random.uniform(1, 3))
    print(f"Потік '{name}': Завершив роботу.")

print("\n--- Приклад 6: Синхронізація з Event ---")
worker1 = threading.Thread(target=worker_thread_event, args=("Робітник
1",))
worker2 = threading.Thread(target=worker_thread_event, args=("Робітник
2",))

worker1.start()
worker2.start()

print("Головний потік: Виконую підготовку...")
time.sleep(2) # Імітація підготовки
print("Головний потік: Підготовка завершена. Подаю сигнал!")
start_event.set() # Встановлюємо подію, розблоковуючи робочі потоки

worker1.join()
worker2.join()
print("Всі потоки Прикладу 6 завершили виконання.")
```

# Завдання до Прикладу 6: Використання `threading.Event`

Створіть "серверний" потік та "клієнтський" потік. Клієнтський потік має чекати, доки серверний потік не буде готовий(наприклад, "Сервер готовий приймати запити"). Після сигналу клієнтський потік надсилає "запит".

Відповідь