Synchronizacja wątków i unikanie zakleszczeń w Pythonie

Wątek jest to jednostka wykonawcza w programie, która może działać równolegle z innymi wątkami. Należy rozróżniać pojęcie procesu, czyli samodzielnej jednostki wykonywalnej z własną przestrzenia pamięci, od pojęcia wątku, który współdzieli pamięć i zasoby z innymi wątkami w tym samym procesie.

Prościej wyjaśniając to **proces** jest pojemnikiem na **wątki** i w jego obrębie działa minimum jeden wątek. Przestrzeń pamięci procesu jest dzielona między wątkami.

Cechy wątku:

- wszystkie wątki w procesie mają dostęp do tej samej pamięci
- ułatwiona komunkacja między wątkami (ryzyko race condition)
- tworzenie i przełączanie między wątkami jest szybsze niż między procesami
- mogą wykonywać różne zadania jednocześnie (efektywnie dla operacji I/O-bound).
- działają niezależnie i mogą zakończyć się, nie wpływając na inne wątki (chyba że są zależności).

Wyróżnić można właściwie trzy grupy wątków:

1. wątki realizowane na poziomie systemu operacyjnego Nazywane również jako wątki z wywłaszczaniem (ang. preemptive), kernel-level, OS-level, native itp.,
2. wątki realizowane na poziomie maszyny wirtualnej, czy np. biblioteki Nazywane również: wątki bez wywłaszczania (ang. non-preemptive), user-level, wątki kooperatywne (ang. cooperative), "zielone" (ang. green)
3. hybrydowe, czyli połączenie powyższych, np. wiele wątków non-preemptive może być uruchomionych w kilku wątkach preemptive,

jednak w Pythonie nie jest to takie proste a wszystko przez **GIL**.

Global Interpreter Lock (GIL) w Pythonie uniemożliwia równoczesne wykonywanie wątków w jednym procesie (dla kodu Pythona).

- Kiedy GIL nie jest problemem:
  - Dla zadań I/O-bound, gdzie GIL jest uwalniany podczas operacji wejścia/wyjścia. (Preemptive)
- Kiedy GIL jest ograniczeniem:
  - Dla zadań CPU-bound. Rozwiązanie: Użycie multiprocessing. (Non-preemptive)

Uwaga!

Dla zadań I/O-bound wątki w Pythonie są bardzo wydajne, ale dla zadań CPU-bound warto rozważyć `multiprocessing`, który postaram się przedstawić na końcowych zajęciach.

1. Wstęp do wątków w Pythonie

Wątki pozwalają na wykonywanie wielu operacji jednocześnie w obrębie jednego programu. Python oferuje obsługę wątków za pomocą modułu `threading`. Ogólnie przyjęto, że praca z wątkami dzieli się na trzy sposoby:

- klasa `Thread`,
- dziedziczenie z klasy `Threads`,
- pula wątków.

Klasa `Thread`

```
from threading import Thread
import threading

def print_numbers():
    for i in range(5):
        print(f"Number: {i}")

thread = Thread(target=print_numbers)  # Tworzenie nowego wątku
thread.start()  # Uruchamianie wątku
thread.join()  # Czekanie na zakończenie wątku
```

```
Number: 0
Number: 1
Number: 2
Number: 3
Number: 4
```

W powyższym kodzie `threading.Thread`, tworzy instancję wcześniej wspomnianej klasy `Thread`. Argument `target` wskazuje funkcję, którą wątek ma wykonać.

- `start()` - rozpoczyna wykonywanie nowego wątku.
- `join()` - sprawia, że program czeka na zakończenie pracy wątku.

Problem związany z wątkami dotyczy używania ich w środowiskach Notebook, ze względu na zapis do danych bufora `stdout`, co skutkuje nieprzewidzianymi działaniami, gdzie wywołanie bloku kodu, może spowodować, dołączenie wyjsciowej zawartości do poprzednich zadań wykorzystujących wątki.

Oczywiście mamy możliwość, zobaczenia jakie wątki są aktualnie wykonywane służy do tego funkcja `threading.enumarate()`.

```
threading.enumerate()
```

```
[<_MainThread(MainThread, started 130352613598720)>,
 <Thread(IOPub, started daemon 130352456664768)>,
 <Heartbeat(Heartbeat, started daemon 130352446179008)>,
 <Thread(Thread-2 (_watch_pipe_fd), started daemon 130352414721728)>,
 <Thread(Thread-3 (_watch_pipe_fd), started daemon 130352404235968)>,
```

```
 <ControlThread(Control, started daemon 130352322447040)>,
 <HistorySavingThread(IPythonHistorySavingThread, started
130352311961280)>,
 <ParentPollerUnix(Thread-1, started daemon 130352301475520)>]
```

1. **`MainThread`**:
   - Główny wątek programu.
   - Wszystkie wątki są tworzone przez ten wątek.
2. **`IOPub`** i **`Heartbeat`**:
   - Wątki wewnętrzne Jupyter Notebook:
     - `IOPub` - odpowiada za przekazywanie danych wyjściowych (stdout, stderr) z kernela do interfejsu użytkownika Jupyter Notebook.
     - `Heartbeat` -sprawdza, czy kernel Jupyter działa poprawnie.
3. **`_watch_pipe_fd`** - wątki monitorujące dane przesyłane między procesami Jupyter.
4. **`Control`** - wątek zarządzający komunikacją między kernelem a klientem (np. notebookiem).
5. **`IPythonHistorySavingThread`** - wątek odpowiedzialny za zapisywanie historii komend IPython w Jupyter Notebook.
6. **`ParentPollerUnix`** - wątek sprawdzający, czy proces nadrzędny (np. Jupyter Notebook) nadal działa. Jeśli nie, kernel się wyłącza.

W praktyce korzystając z przykładowego IDE (PyCharm) jedyny wątkek, jaki powinien być widoczny to przed przystąpieniem do pracy z wątkami to **`MainThread`**.

`daemon` w Pythonie, to typ wątku, który działa w tle i automatycznie kończy się się gdy wszystkie wątki "nie-daemon" zakończą swoje działanie. Zazwyczaj są używane do wykonywania zadań pomocniczych, które nie są krytyczne dla działania aplikacji.

Jeżeli chodzi o różnicę w działaniu to rozważmy dwa przypadki, proszę użyć pełnoprawnego IDE.

1. Tworzony tworzony jest zwykły wątek

```python
from threading import Thread
import time

def non_daemon_task():
    time.sleep(5)
    print("Non-daemon thread finished.")

thread = Thread(target=non_daemon_task)
thread.start()

print("Main thread is done.")
```

```
Main thread is done.
Daemon thread finished.
```

1. Tworzony jest wątek pomocniczy `deamon` (przy pomocy odpowiedniej flagi `.daemon=True`)

```python
from threading import Thread
import time

def daemon_task():
    time.sleep(5)
    print("Daemon thread finished.")  # To się nie wyświetli

thread = Thread(target=daemon_task)
thread.daemon = True
thread.start()

print("Main thread is done.")
```

W rezultacie działania:

1. Wątek nie-daemon będzie działać przez 5 sekund, zanim się zakończy, ponieważ Python czeka na jego zakończenie
2. Wątek daemon zostanie przerwany i jego kod (print) nigdy się nie wykona.

Zastosowanie podejścia tego typu wątków najczęściej występuje w:

- regularnym odświeżaniu danych, np. synchronizacja czasu systemowego, pobieranie danych z API
- nasłuchiwaniu połączeń sieciowych w tle.
- zbieraniu logów, monitorowaniu działania systemu lub aplikacji w tle
- obsłudze aplikacji graficznej, np. aktualizacja elementów GUI w tle.

Ogólnie przyjęto, że stosuje się je tam, gdzie nie wymaga się dokładnego zarządzania.

Dziedziczenie po `Thread`

```python
from threading import Thread

class MyThread(Thread):
    def __init__(self, i):
        super().__init__()
        self._i = i

    def run(self) -> None:
        print(self._i)


threads = []

for i in range(20):
```

```
    threads.append(MyThread(i=i))

for t in threads:
    t.start()

for t in threads:
    t.join()

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
```

W powyższym kodzie, klasa `MyThread` dziedziczy po klasie `Thread`. Podobnie jak w laboratoriach poświęconych obiektowości, tutaj `super().__init__()` wywołuje konstruktor klasy nadrzędnej (threading.Thread), aby prawidłowo zainicjalizować mechanizmy wątkowe.

W rezultacie działania przedstawionego kodu, każde wyświetlenie odbywa się niezależnie i dotyczy osobnych wątków.

Pula wątków `ThreadPoolExecutor`

`ThreadPoolExecutor` jest częścią modułu `concurrent.futures`, który dostarcza wysokopoziome API do zarządzania równoległością w wielu wątkach w sposób prosty i uporządkowany. Zasada działania jest bardzo podobna do tworzenia puli połączeń z bazą danych, tak jak to było w przypadku `MySQL`.

```
from concurrent.futures import ThreadPoolExecutor

with ThreadPoolExecutor(max_workers=20) as executor:
    for i in range(20):
        executor.submit(print, i)
```

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
```

- `ThreadPoolExecutor` jest narzędziem, które umożliwia wykonywanie zadań równolegle.
- Funkcja `submit(print, i)` dodaje zadanie do puli wątków, które jest wykonywane równolegle.

2. Problemy współbieżności

Przy współdzieleniu zasobów przez wątki może dojść do problemów, takich jak **race conditions**.

Race condition to sytuacja, która występuje w programowaniu współbieżnym, gdy dwa lub więcej wątków lub procesów współbieżnie próbuje uzyskać dostęp do tego samego zasobu (np. zmiennej, pliku, bazy danych) i przynajmniej jeden z nich modyfikuje ten zasób. Jeśli dostęp i modyfikacja nie są odpowiednio zsynchronizowane, końcowy stan zasobu może być nieprzewidywalny i zależy od kolejności wykonania wątków.

```python
import threading
import time

counter = 0  # Zmienna współdzielona

def increment_counter():
    global counter
    for _ in range(1000):
        current = counter  # Odczyt wartości
        time.sleep(0.0001)  # Symulacja opóźnienia
        counter = current + 1  # Aktualizacja wartości

# Tworzenie i uruchamianie 10 wątków
```

```
threads = [threading.Thread(target=increment_counter) for _ in
range(10)]
for thread in threads:
    thread.start()
for thread in threads:
    thread.join()

print(f"Final counter value: {counter}")

Final counter value: 1003
```

Powyższy przykład, to typowa symulacja mająca pokazać problem **race conditions**. Wątki bez określonej synchronizacji starają się przeprowadzić operację, uzyskiwania dostępu do zmiennej globalnej `counter`. Funkcja `time.sleep()`, została użyta w celu wymuszenia pewnego opóźnienia, które często towarzyszy podczas działania programów (przykładowo żadanie HTTP). Przez to wątki uzyskują dostęp do niezakutalizowanej wartości `counter` i wyniki różnią się od tych, które powinny zostać zwrócone.

`global counter` oznacza, że zmienna counter, używana w funkcji `increment_counter`, odnosi się do globalnej zmiennej `counter` zdefiniowanej na początku programu.

W poniższym kodzie, natomiast jest przedstawiony bardziej realny przypadek, gdzie występuje naturalne opóźnienie spowodowane komunikacją z systemem plików. Uruchamia on 10 wątków, każdy wątek ma odczytać plik, pobrać jego zawartość, dodać do niej 1 i zapisać ją do pliku. Zawartość pliku na początku wynosi 0. Prostym jest, że oczekujemy na końcu działania, że wartość będzie równa 10.

```
from threading import Thread
from concurrent.futures import ThreadPoolExecutor

variable: int = 1
FILENAME: str = 'test'


def read_from_file() -> int:
    with open(FILENAME, 'r+') as file:
        val = file.read()
        return int(val)


def write_to_file(val: int) -> None:
    with open(FILENAME, 'w') as file:
        file.write(str(val))


def increment_value_in_file():
    value = read_from_file()
    write_to_file(value + 1)
```

```
write_to_file(0)


with ThreadPoolExecutor(10) as executor:
    for _ in range(10):
        executor.submit(increment_value_in_file)

print("Final value: ", read_from_file())
```

```
Final value:  2
```

Uzyskany wynik jest niedeterministyczny, ponieważ problemem jest brak synchronizacji. Wątki przeplatają się odczytując i zapisując plik, co sprawia, że nic nie stoi na przeszkodzie by np. trzy wątki odczytały tę samą wartość z pliku, bo żaden inny wątek nie zdąży nic zapisać. To natomiast sprawi, że te trzy wątki będą próbowały zapisać tę samą wartość do pliku.

3. Synchronizacja wątków

Aby zapobiec problemom współbieżności, możemy użyć mechanizmów synchronizacji.

`Lock()` (blokada) w Pythonie to mechanizm synchronizacji, który zapobiega jednoczesnemu dostępowi wielu wątków do współdzielonego zasobu (np. zmiennej, pliku czy bazy danych). Lock gwarantuje, że tylko jeden wątek na raz może uzyskać dostęp do krytycznej sekcji kodu, co pozwala uniknąć problemu race condition. Stosowanie `Lock`, często jest powodem problemu `DeadLock`.

```
def increment_counter():
    global counter
    for _ in range(1000):
        lock.acquire()  # Ręczne zdobycie blokady
        try:
            current = counter  # Odczyt wartości
            time.sleep(0.0001)  # Symulacja opóźnienia
            counter = current + 1  # Aktualizacja wartości
        finally:
            lock.release()  # Ręczne zwolnienie blokady

import threading
import time

counter = 0  # Zmienna współdzielona
lock = threading.Lock()  # Blokada do synchronizacji

def increment_counter():
    global counter
    for _ in range(1000):
        with lock:  # Sekcja krytyczna chroniona blokadą
            current = counter  # Odczyt wartości
            time.sleep(0.0001)  # Symulacja opóźnienia
            counter = current + 1  # Aktualizacja wartości
```

```
# Tworzenie 10 wątków
threads = [threading.Thread(target=increment_counter) for _ in
range(10)]

# Uruchamianie wątków
for thread in threads:
    thread.start()

# Czekanie na zakończenie wszystkich wątków
for thread in threads:
    thread.join()

# Wyświetlenie końcowej wartości licznika
print(f"Final counter value (with lock): {counter}")
```

```
Final counter value (with lock): 10000
```

Reentrant Lock (RLock) to specjalna wersja blokady (Lock) w Pythonie, która pozwala wątkowi wielokrotnie zdobywać tę samą blokadę bez ryzyka zakleszczenia `deadlock`. Jest to możliwe dzięki temu, że blokada śledzi, ile razy została zdobyta przez ten sam wątek i wymaga tyle samo zwolnień `release`, aby była dostępna dla innych wątków.

```
rlock = threading.RLock()

def nested_locks():
    with rlock:
        with rlock:
            print("Reentrant lock acquired twice")

thread = threading.Thread(target=nested_locks)
thread.start()
thread.join()
```

```
Reentrant lock acquired twice
```

4. Zakleszczenia (Deadlock)

Zakleszczenie występuje, gdy dwa (lub więcej) wątki czekają na zasoby blokowane przez siebie nawzajem.

```
import threading
import time

# Tworzenie dwóch blokad
lock1 = threading.Lock()
lock2 = threading.Lock()

def thread1():
```

```
    with lock1:
        print("Thread 1 acquired lock1")
        time.sleep(0.1)  # Symulacja opóźnienia
        with lock2:  # Czeka na lock2, który może być zablokowany
przez thread2
            print("Thread 1 acquired lock2")

def thread2():
    with lock2:
        print("Thread 2 acquired lock2")
        time.sleep(0.1)  # Symulacja opóźnienia
        with lock1:  # Czeka na lock1, który może być zablokowany
przez thread1
            print("Thread 2 acquired lock1")

# Tworzenie i uruchamianie wątków
t1 = threading.Thread(target=thread1)
t2 = threading.Thread(target=thread2)

t1.start()
t2.start()

# Czekanie na zakończenie wątków (nigdy się nie zakończą w przypadku
zakleszczenia)
t1.join()
t2.join()
```

```
Thread 1 acquired lock1
Thread 2 acquired lock2
```

```
---------------------------------------------------------------------------
-----
KeyboardInterrupt                         Traceback (most recent call
last)
Cell In[205], line 30
     27 t2.start()
     29 # Czekanie na zakończenie wątków (nigdy się nie zakończą w
przypadku zakleszczenia)
---> 30 t1.join()
     31 t2.join()

File ~/anaconda3/lib/python3.12/threading.py:1149, in
Thread.join(self, timeout)
   1146     raise RuntimeError("cannot join current thread")
   1148 if timeout is None:
-> 1149     self._wait_for_tstate_lock()
   1150 else:
   1151     # the behavior of a negative timeout isn't documented, but
   1152     # historically .join(timeout=x) for x<0 has acted as if
timeout=0
```

```
   1153     self._wait_for_tstate_lock(timeout=max(timeout, 0))

File ~/anaconda3/lib/python3.12/threading.py:1169, in
Thread._wait_for_tstate_lock(self, block, timeout)
   1166     return
   1168 try:
-> 1169     if lock.acquire(block, timeout):
   1170         lock.release()
   1171         self._stop()

KeyboardInterrupt:
```

5. Unikanie zakleszczeń

Zasady hierarchii blokad:

- W tym przypadku jest to, zależne od implementacji. To znaczy, że odgórnie należy okreslić hierarchię wątków i wykonywać je w okreslonej kolejności.

Używanie timeout w Lock

- Drugi sposób polega, na celowym oczekiwaniu na dostęp w przypadku gdy blokada jest zajęta. W przypadku przekroczenia czasu oczekiwania metoda `.acquire()` zwraca `False`. Ten mechanizm pozwala na uniknięcie problemu `DeadLock`.

```python
import threading
import time

# Tworzenie dwóch blokad
lock1 = threading.Lock()
lock2 = threading.Lock()

# Funkcja z potencjalnym zakleszczeniem
def thread1():
    if lock1.acquire(timeout=1):
        try:
            print("Thread 1 acquired lock1")
            time.sleep(0.1)  # Symulacja opóźnienia
            if lock2.acquire(timeout=1):
                try:
                    print("Thread 1 acquired lock2")
                    print("Thread 1 is in critical section")
                finally:
                    lock2.release()
            else:
                print("Thread 1 failed to acquire lock2, avoiding
deadlock")
        finally:
            lock1.release()
    else:
```

```
        print("Thread 1 failed to acquire lock1, avoiding deadlock")

def thread2():
    if lock2.acquire(timeout=1):
        try:
            print("Thread 2 acquired lock2")
            time.sleep(0.1)  # Symulacja opóźnienia
            if lock1.acquire(timeout=1):
                try:
                    print("Thread 2 acquired lock1")
                    print("Thread 2 is in critical section")
                finally:
                    lock1.release()
            else:
                print("Thread 2 failed to acquire lock1, avoiding
deadlock")
        finally:
            lock2.release()
    else:
        print("Thread 2 failed to acquire lock2, avoiding deadlock")

# Tworzenie i uruchamianie wątków
t1 = threading.Thread(target=thread1)
t2 = threading.Thread(target=thread2)

t1.start()
t2.start()

# Czekanie na zakończenie wątków
t1.join()
t2.join()

print("Program completed")

Thread 1 acquired lock1
Thread 2 acquired lock2
Thread 1 failed to acquire lock2, avoiding deadlock
Thread 2 acquired lock1
Thread 2 is in critical section
Program completed
```

6. Alternatywy: Semaphore i Condition

W Pythonie **`Semaphore`* jest kolejnym mechanizmem synchronizacji, który ogranicza liczbę wątków, które mogą jednocześnie uzyskać dostęp do zasobu lub sekcji krytycznej. W poniższym kodzie `threading.Semaphore(3)` oznacza, że maksymalnie 3 wątki mogą jednocześnie wykonywać sekcję chronioną przez semafor.

- `threading.Event().wait(2)` to metoda używana do wprowadzenia kontrolowanego opóźnienia w działaniu wątku i sprawia, że wątek czeka 2 sekundy zanim

wznowi swoje działanie. Ma przewagę nad `time.sleep(2)`, bo daje możliwość wcześniejszego wykonania wątku.

```
semaphore = threading.Semaphore(3)

def limited_access():
    with semaphore:
        print(f"Accessing resource:
{threading.current_thread().name}")
        threading.Event().wait(2)


threads = [threading.Thread(target=limited_access) for _ in range(10)]
for thread in threads:
    thread.start()
for thread in threads:
    thread.join()
```

```
Accessing resource: Thread-2006 (limited_access)
Accessing resource: Thread-2007 (limited_access)
Accessing resource: Thread-2008 (limited_access)
Accessing resource: Thread-2009 (limited_access)
Accessing resource: Thread-2011 (limited_access)
Accessing resource: Thread-2010 (limited_access)
Accessing resource: Thread-2012 (limited_access)
Accessing resource: Thread-2013 (limited_access)
Accessing resource: Thread-2014 (limited_access)
Accessing resource: Thread-2015 (limited_access)
```

Condition umożliwia synchronizację z warunkami (np. oczekiwanie na zmianę stanu). Przykładowo w poniższym kodzie służy on do synchronizacji między producentem a konsumentem, zapewniając, że konsument nie spróbuje użyć zasobu, zanim zostanie on wyprodukowany.

- `condition.wait()`
  - Konsument czeka, aż producent zasygnalizuje dostępność zasobu.
  - Wątek zostaje wstrzymany do momentu, gdy inny wątek wywoła `condition.notify()`.
- `condition.notify()`
  - Producent sygnalizuje konsumentowi, że zasób został utworzony.
  - Konsument zostaje obudzony i może kontynuować swoje działanie.
- Blokada w `Condition` - działa w połączeniu z wewnętrzną blokadą (`Lock`), która automatycznie zarządza dostępem do współdzielonego zasobu.

```
condition = threading.Condition()
shared_resource = []

def producer():
    with condition:
        shared_resource.append(1)
```

```
        print("Produced an item")
        condition.notify()

def consumer():
    with condition:
        while not shared_resource:
            condition.wait()
        print("Consumed an item", shared_resource.pop())

threading.Thread(target=producer).start()
threading.Thread(target=consumer).start()

Produced an item
Consumed an item 1
```

Zadanie 1. Symulacja bankomatu

Stwórz program (funckja lub klasa) symulujący działanie bankomatu, w którym kilka klientów (wątków) próbuje równocześnie wypłacać pieniądze z jednego konta bankowego. Użyj mechanizmu synchronizacji, aby upewnić się, że saldo konta nigdy nie spadnie poniżej zera (rozważ sytuację, że w przypadku tak delikatnych operacji należy przewidzieć problem race condition i zakleszczeń).

1. Początkowe saldo: 100 zł.
2. Klienci wypłacają losowe kwoty (10-50 zł).
3. Funkcja `withdraw(client_id)`:
   - Sprawdza dostępne saldo.
   - Jeśli możliwe, zmniejsza saldo i wypisuje komunikat.
   - W przeciwnym razie wypisuje komunikat o braku środków.

Zadanie 2. Praca nad projektem

Przeanalizuj swój projekt pod kątem możliwości wykorzystania wątków do poprawy wydajności lub równoczesnego przetwarzania zadań. Zastanów się, które elementy projektu można wykonać współbieżnie, na przykład:

- Obsługa wielu użytkowników lub procesów jednocześnie.
  - Równoczesne uruchamianie wielu kontenerów
  - Równoczesne monitorowanie wielu zadań konserwatora
- Rozdzielenie zadań intensywnych obliczeniowo (np. przetwarzanie danych, obliczenia, detekcja obiektów).
  - Przetwarzanie obrazów równolegle (np. detekcja na kilku obrazach jednocześnie).
- Wykonywanie operacji w tle (np. zapis do bazy, komunikacja z serwerem, analiza danych).
  - Jednoczesne pobieranie danych pogodowych z różnych źródeł.
  - Obsługa gry w tle (np. timer w grze, regeneracja życia postaci)