**Wielowątkowość**

1. **Czym jest wielowątkowość i multiprzetwarzanie?**

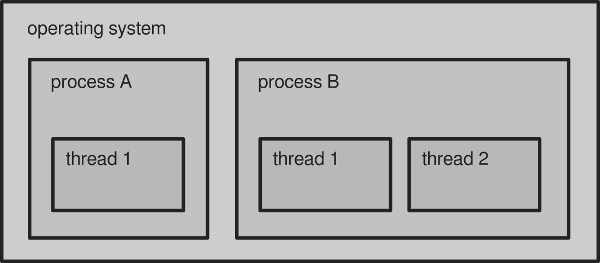
W trakcie pracy na komputerze nie wyobrażamy sobie, abyśmy nie mieli możliwości uruchamiania wielu programów równocześnie. Założę się, że nierzadko robisz kilka czynności równocześnie, np. czytasz to szkolenie, w tle słuchasz muzyki i w kolejnym oknie masz uruchomionego Facebook’a.

Podchodząc do tego bardziej technicznie - każdy program to tak naprawdę **oddzielny proces** uruchamiany na odpowiednich **rdzeniach procesora**.

Każdy z tych procesów składa się z **wątków**.

Wątki realizują podzadania danego programu/procesu (np. abyś mógł słuchać muzyki z Spotify, konieczne jest, aby poprawnie działał wątek odpowiedzialny za transfer danych między serwerem a aplikacją oraz inny wątek służący do obsługiwania głośnika komputera).   
Takie właśnie funkcjonowanie i współpracowanie ze sobą wielu wątków równocześnie w ramach jednego procesu, nazywamy multithreadingiem (wielowątkowością).

Każdy proces **musi posiadać chociaż jeden wątek**, który jest uruchamiany na początku działania programu i nazywamy go **wątkiem głównym**.



**Podział na procesy i wątki**

1. **Różnica między procesem a wątkiem**

Zasadnicza i główna różnica między tymi dwoma tworami sprowadza się do tego, że każdy proces ma przydzielony **na własność pewien obszar pamięci**. Domyślnie obszar ten (choć można to zmienić dzięki technikom **inter-process communication - IPC**) nie może być współdzielony z innym procesem. I tak na przykład zasoby aplikacji, z której słuchasz muzyki nie mogą być wykorzystywane przez przeglądarkę internetową i vice versa.

Natomiast wątki należące do jednego procesu **współdzielą między sobą zasoby** i mogą wymieniać informacje. Z racji, że wątki wchodzą w skład danego procesu, są one nazywane również **lekkimi procesami** (ang. lightweight processes).

Operacje jakie realizowane są w ramach pojedynczego programu, rozkładane są na poszczególne wątki i dzięki temu możemy otrzymywać **efekt współbieżnego** działania uruchomionych programów.

W przeciwnym razie, gdyby wątki nie istniały, każda pojedyncza instrukcja aplikacji musiałaby być uruchamiana jako odrębny proces, co znacznie utrudniłoby pracę z zasobami.

1. **Współbieżność vs równoległość**

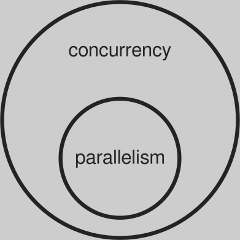
Tak jak już wspomniałem, proces jest podzielony na wątki, aby możliwe było przyspieszenie działania programu. Dzięki temu teoretycznie mamy możliwość rozłożenia funkcjonalności procesu w taki sposób, że operacje programu będą iluzorycznie wykonywały się równolegle (dlaczego iluzorycznie? czytaj dalej!)   
Wyobraź sobie, że program renderujący film może tak podzielić dane kawałki filmu, że jego poszczególne części będą przetwarzane przez różne wątki.

Dzięki temu, w momencie gdy standardowe renderowanie filmu zajęłoby 1 h, to dzieląc program na 2 równoległe wątki, renderowanie **mogłoby** zająć 30 minut. Mówię mogłoby, bo w praktyce nie jest tak kolorowo.

Musisz bowiem wiedzieć, że nigdy nie masz 100% gwarancji, że wątki będą wykonywane równolegle w tym samym czasie.

Procesor z pewnych ograniczeń często **pozoruje zrównoleglenie** operacji.

Wtedy mówimy o tzw. **współbieżności (ang. concurrency)**. Może to wydawać się nieco mylące, ale musisz wiedzieć, że **równoległość to nie współbieżność**! To, czy program faktycznie wykona się w pełni równolegle zależy od niskopoziomowych czynników architektury procesora, nad którymi schylimy się za chwilę.



**Współbieżność vs równoległość**

1. **Kiedy faktycznie mamy do czynienia z równoległością?**

Aby procesor faktycznie był w stanie wykonywać czynności w pełni równolegle, musi on posiadać więcej niż jeden rdzeń ([**rdzeń fizyczny**](https://pl.wikipedia.org/wiki/Rdze%C5%84_procesora) lub [**wirtualny**](https://forum.pcformat.pl/Rdzenie-Wirtualne-t)).

Jak się okazuje bowiem, rdzeń procesora jest w stanie przetwarzać tylko **jedną operację równocześnie**.

Nie oznacza to jednak, że na jednym rdzeniu CPU może być uruchomiony tylko jeden proces. Na jednym rdzeniu fizycznym możemy odpalać większą ilość programów, ale ich wątki wykonywane będą **współbieżnie**, **nie równolegle**.

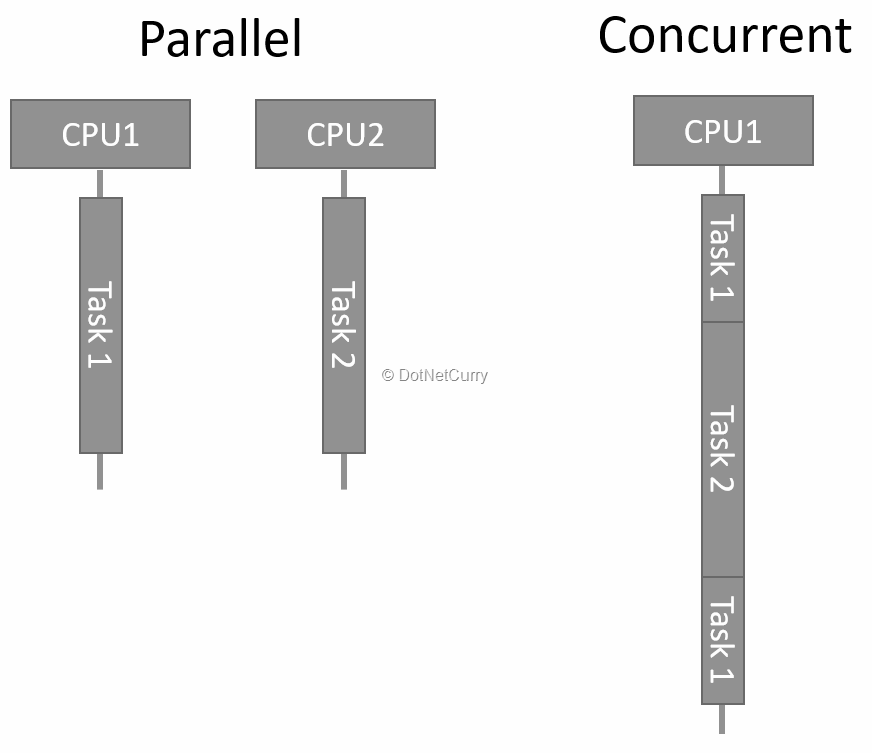
Współbieżność wykonywania polega na uruchamianiu każdego wątku w pętli na określony czas. Jądro systemu kontroluje, ile tzw. czasu procesora ma zostać przydzielone na wykonanie wątku w pojedynczym cyklu. Powiem więcej - mimo że, dane wątki różnych procesów działają na jednym rdzeniu w pętli, to i tak użytkownik ma wrażenie ich równoległego działania.

Wynika to z tego, że przydzielany czas procesora na wykonanie operacji, jest na tyle krótki, że układ nerwowy człowieka nie potrafi dostrzec jakichkolwiek przeskoków między wątkami.

Dla lepszego zobrazowania, mając uruchomioną na tym samym rdzeniu, np. aplikację odtwarzającą muzykę i inną - wyświetlającą strony internetowe, następuje tak szybkie przeskakiwanie między wykonywaniem tych programów, że użytkownik odnosi wrażenie, jakby **wykonywały się one równolegle** (a wykonują się tak naprawdę współbieżnie).

Tak więc może to wyglądać następująco (zakładając, że procesy uruchomione są na jednym rdzeniu):

* w pojedynczym cyklu procesora, 100 ms wykonuje się aplikacja od muzyki, po tym czasie jest ona wstrzymywana i na kolejne 100 ms uruchomiona jest aplikacja wyświetlająca strony internetowe. I tak w kółko. Przez to, że czas nieustannego wykonywania tych pojedynczych programów jest tak krótki, użytkownik ma wrażenie, jakby procesy te wykonywały się faktycznie w pełni równolegle.



**UWAGA**:

CPU1, CPU2 to rdzenie procesora.

1. **GIL w Pythonie**

Z tego miejsca muszę wspomnieć jeszcze o tzw. **Global Interpreter Lock** jako jednej z cech Pythona.

W skrócie - GIL - uniemożliwia równoległe wykonywanie kilku wątków równocześnie (nawet gdy uruchomione są one na kilku rdzeniach procesora).

Wprowadzenie GILu wynika z faktów historycznych, kiedy to mieliśmy tylko do czynienia z jednowątkowymi programami.

Zapytasz więc, dlaczego nie zdecydowano się na usunięcie lub zastąpienie GILa jakimś innym feature-m.

Otóż GIL faktycznie nakłada pewne ograniczenia na wielowątkowe programy wykonywane w ramach jednego interpretera, ale drugą stroną medalu jest fakt, iż GIL przyspiesza czas wykonywania programu jednowątkowego.

Właśnie ten fakt jest zapewne głównym powodem, dlaczego GIL ma swoje miejsce również w nowszych standardach języka.

Istnieją również sposoby na ominięcie wspomnianego ograniczenia i chcąc faktycznie korzystać z zalet r**ównoległego wykonywania operacji**, możliwe jest **utworzenie kilku procesów** (już nie kilku wątków, które ogranicza GIL) i to na nie przekierować zrównoleglenie operacji.

Tak więc chciałbym, żebyś z tego podpunktu wyciągnął jeden najważniejszy wniosek. Utworzenie programu wielowątkowego w Pythonie wcale nie przyspieszy wykonywania jakichś złożonych operacji przez procesor.

Środowisko bazujące na kilku współbieżnych wątkach powinniśmy budować w przypadku **tworzenia programów I/O-bound** (lub zastosować korutyny z modułem asyncio, o którym w innych szkoleniach).

Programy I/O-bound wykonują operacje pokroju:

* czekanie na pobranie danych z jakiegoś serwera, oczekiwanie na wprowadzenie danych przez użytkownika itp.

Tak więc uruchamiając je współbieżnie, wcale nie zależy Nam na przyspieszeniu operacji (dane i tak będą się pobierały przez określony czas zależny z góry od szybkości łącza internetowego użytkownika, a nie wydajności CPU), ale otrzymania wrażenia **równoległego przetwarzania danych** w tle.

1. **Problemy wielowątkowego przetwarzania**

Jak już wiemy, wątki mogą dzielić ten sam obszar pamięci. Dzięki temu możemy łatwo integrować ze sobą działanie kilku wątków działających w obrębie jednego procesu.

Problem jednak pojawia się w momencie, gdy dwa wątki chcą skorzystać z tego samego zasobu pamięci w tym samym czasie. Przykładem może być sytuacja, gdy jeden wątek **odczytuje dane z pamięci**, a drugi równocześnie **coś do niej zapisuje**.

Wówczas mówimy o problemach:

* **Data race** - występuje, gdy pierwszy wątek modyfikuje pewien obszar pamięci, a drugi chce równocześnie coś z niego odczytać.   
  Jeżeli wątek zapisujący nie zakończy swojej pracy przed rozpoczęciem procesu czytania, to zostaną pobrane naruszone dane.
* **Race condition** występuje, gdy wątki wykonują na danym zasobie pamięci operacje w nieustalonej kolejności. Chodzi o przypadek, gdy, np. pewien wątek **pobierający informacje z bazy danych** będzie wykonywał się przed wątkiem, **przetwarzającym te dane**.   
  Wówczas kolejność operacji będzie nieprawidłowa (jak mają zostać przetworzone dane, gdy nie zostały jeszcze w pełni pobrane z bazy?).

Dlatego, aby zabezpieczyć się przed wyżej wymienionymi zagrożeniami i niepożądanymi działaniami, musimy tworzyć programy, tzw. **thread safe** lub upewnić się, że korzystamy z bibliotek, które właśnie to bezpieczeństwo zapewniają.

1. **Rozwiązanie powyższych problemów**

Zarówno data-race jak i race condition to rzeczywiste problemy w technologii informatycznej. Niegdyś, gdy efektywność wielowątkowości była dość prymitywna, stanowiła ona poważne zagrożenie dla życia ludzi.   
Nieraz mówi się o Therac-25, czyli maszynie służącej do radioterapii, której to właśnie błędy oprogramowania wynikające z niewłaściwego działania procesów spowodowały śmierć kilku pacjentów.

Obecnie możliwymi rozwiązaniami dla powyższych problemów są:

* **synchronizacja** - synchronizując wykonanie kilku wątków, zapewniamy, że tylko jeden będzie wykonywał się w danym czasie. Dzięki temu zabezpieczamy się przed równoczesnym operowaniem na danym zasobie pamięci.
* **operacje atomowe** operacje będące potencjalnym zagrożeniem dla bezpieczeństwa przetwarzania danych, możemy czasami ustawić jako atomowe, przez co dane będą niepodzielne i niemożliwe będzie ich naruszenie przez 2 wątki równocześnie
* **dane niemutowalne** zagrożenie w przetwarzaniu wielowątkowym jest niwelowane w momencie, gdy wątki mogą tylko czytać dane, bez możliwości ich modyfikowania.   
  Nie wiem, czy pamiętasz, ale zapewnianie niemutowalności danych jest główną filozofią programowania funkcyjnego. Dlatego też często mówimy o tym paradygmacie jako wątkowo-bezpiecznym.

1. **Wielowątkowość w Pythonie**

W Pythonie mamy do dyspozycji dwa moduły obsługujące wielowątkowość - tzw. moduł **thread** oraz **threading**. Aczkolwiek pierwszy z nich został uznany za przestarzały i obecnie zaleca się używanie **modułu threading**.

Dzieląc wykonanie programu na wątki, możliwe jest odpalenie “w tle” oddzielnego wątku pobierającego dane z API z serwera, nie blokując równocześnie działania całej aplikacji.

**Utworzenie i uruchomienie pojedynczego wątku**

| import threading  def new\_thread():  print("Thread is running...")  for x in range(10):  print(x)  t1=Thread(target=new\_thread)  t1.start()  t1.join() |
| --- |

**Wyjaśnienie:**

Aby utworzyć instancję wątku, użyjemy konstruktora **Thread()** wraz z argumentem **target**, do którego będziemy przypisywali referencję funkcji, która ma być uruchamiana w ramach wątku.

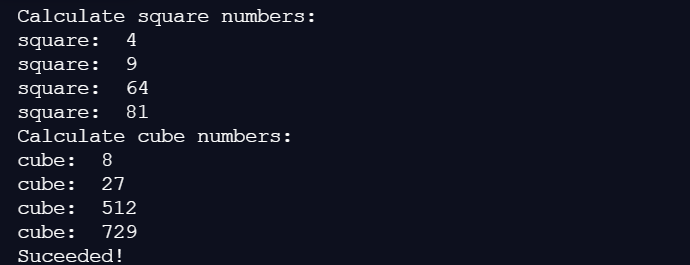
Następnie, aby prawidłowo uruchomić wątek, użyjemy metody **.start()**. **Join()** natomiast jest konieczny do użycia, aby zapewnić prawidłowe wykonanie całego programu. Metoda ta bowiem “dołącza” utworzony wątek do programu (wątku głównego) i dzięki temu program nie wyłączy się, dopóki utworzony wątek nie zakończy swojej pracy.

1. **Pierwszy program wielowątkowy**

**Przykład**: *Stwórz 2 współbieżne wątki - pierwszy ma wyświetlać kwadraty liczb z listy, drugi natomiast ich wartości sześcienne.*

| **import threading  def calc\_square(numbers):  print("Calculate square numbers: ")  for i in numbers:  print('square: ', str(i\*\*2))  def calc\_cube(numbers):  print("Calculate cube numbers: ")  for i in numbers:  print('cube: ', str(i\*\*3))  def main():  arr = [2,3,8,9]  t1 = threading.Thread(target=calc\_square, args=[arr])  t2 = threading.Thread(target=calc\_cube, args=[arr])   t1.start()  t2.start()  t1.join()  t2.join()   print("Succeeded!")  if \_\_name\_\_ == "\_\_main\_\_":  main()** |
| --- |

**Wynik:**

****

Wyjaśnienie:

1. Powyżej stworzyliśmy program składający się z 2 współbieżnie wykonujących się wątków działających w obrębie jednego procesu **calc\_square** oraz **calc\_cube**.

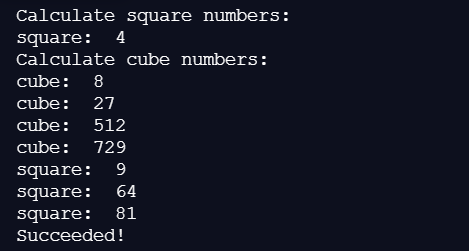
Zwróć uwagę na sposób utworzenia instancji danego obiektu: **threading.Thread(target=calc\_cube, args=[arr])**

Chcąc ustalić w ramach jakiej funkcji wykonywać się będzie tworzony wątek, musimy przekazać mu referencję do tejże funkcji w postaci parametru **target**. Dodatkowo, w celu określenia argumentów przesłanych do wątka, używamy **args**.

Funkcja calc\_square po zmianie:

| import threading import time  def calc\_square(numbers):  print("Calculate square numbers: ")  for i in numbers:  print('square: ', str(i\*\*2))  time.sleep(0.1) # for the sake of better understanding the time slicing |
| --- |
|  |

**Wynik:**



Dzięki dodaniu funkcji usypiającej wątek, widzimy w bardziej wyraźny sposób, jak realizowana jest współbieżność wątkowa w obrębie jednego procesu. Usypiając wątek na 0.1 s, nie mamy już do czynienia z równomiernym time slice-ingiem między wątkami. Zwróć również, że w wyniku współbieżności, uśpienie wątka nie spowodowało wstrzymania całego flow programu.

1. **Data Race**

W początkowej sekcji wspomnieliśmy o problemach, jakie mogą pojawiać się przy pisaniu programów wielowątkowych. Jednym z nich była kłopotliwa sytuacja, w trakcie kilka wątków mogło **dzielić te same zasoby** i je **modyfikować.** Powstawało wtedy zjawisko data race, które może prowadzić do otrzymywania **nieoczekiwanych rezultatów** w programie.

**Przykład:** *Data race na zmiennej typu int - value.*

| import threading  value = 0  def increase\_by\_one():  global value  for i in range(100000):  value += 1  def main():  threads = [threading.Thread(target=increase\_by\_one), threading.Thread(target=increase\_by\_one)]  for thread in threads:  thread.start()   for thread in threads:  thread.join()   print(value) # should be 200000  if \_\_name\_\_ == "\_\_main\_\_":  main() |
| --- |

Wynik:



**Wyjaśnienie:**

Powyższy przykład to chyba mój ulubiony jeżeli chodzi o proste zobrazowanie problemu data race. Zauważ najważniejszą rzecz - wynik (czyli wartość zmiennej value), którego oczekujemy powinien być równy wartości 200000 (uruchomiliśmy bowiem dwa wątki, z których każdy wykonuje oddzielnie 100000 iteracji i zwiększa zmienną value o 1).

Natomiast zmienna value po zakończonych obliczeniach jest równa wartości 167034 (gdzie się podziała pozostała wartość 32966?).

**Dlaczego?**

Problem nieprawidłowego działania powyższego przykładu wynika z przemiennego wykonywania się wątków w cyklach procesora (jako następstwo time slice’ingu) i modyfikowania przez **dwa wątki jednej tej samej zmiennej**.

Takie przeskakiwanie między wątkami może doprowadzić do **niezrealizowania** całej operacji powiększania zmiennej value o wartość 1 i przerwania tego procesu w dowolnym momencie (value += 1).

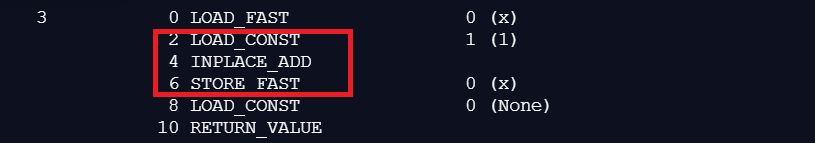
System bowiem może wstrzymać działanie jednego wątku i uruchomić drugi w momencie, gdy procesor jest w trakcie dodawania wartości 1 do value.

Brzmi absurdalnie, ale musisz zdać sobie sprawę z tego, że zwiększanie dowolnej zmiennej o jakieś wartości, z perspektywy CPU podzielone jest na **wiele poszczególnych kroków**, których poprawne działanie może zostać zagrożone i przez nieumiejętnie napisany program wielowątkowy.

**Przykład:** *Kroki procesora przy zwiększaniu wartości zmiennej x o 1 w funkcji.*

| import dis  def inc(x):  x += 1  dis.dis(inc) |
| --- |

**Wynik:**



Zaznaczone na czerwono kroki są niezbędne, by przeprowadzić proces inkrementacji dowolnej zmiennej o 1.

Teraz wyobraź sobie, że przeskok między wykonaniem dwóch wątków nastąpił podczas realizacji **INPLACE\_ADD**. Procesor przez to nie zdążył zwiększyć w danej iteracji wartości value o 1, co w rezultacie powoduje utracenie jednokrotnego zwiększenia zmiennej.

Jak więc możesz zauważyć z przykładu ukazującego problem data race, takie przerwanie procesorowi wykonywania operacji zwiększania wartości, nastąpiło 32966 razy i zaskutkowało otrzymaniem zmiennej równej 167034, a nie 200000.

1. **Zapobieganie data race. Przykład synchronizacji.**

Najpopularniejszym rozwiązaniem zjawiska data race jest użycie, tzw. **lock-a** (nazywanego w innych językach programowania mutex-em).

Lock działa jak zabezpieczenie danego obszaru kodu przed równoczesnym wykonywaniem przez dwa wątki.   
Tak więc umożliwia on jednemu wątkowi **blokowanie dostępu do danej sekcji kodu**. Dopóki obszar ten nie zostanie odblokowany, **nie ma możliwości współbieżnego** przetwarzania tych samych danych.

Proces blokowania/odblokowywania lock-a realizowany jest na metodach **.acquire()** oraz **.release()**. Tylko jeden wątek może zablokować w jednym momencie daną funkcjonalność. W momencie, gdyby chciał się do dostać do zablokowanej strefy inny wątek, to jest on wstrzymywany dopóki wątek główny nie uwolni odblokowanego obszaru.

Przykład: *Użycie lock wraz z metodami acquire oraz release.*

| import threading  value = 0 lock = threading.Lock()  def increase\_by\_one():  global value   lock.acquire()  for i in range(100000):  value += 1  lock.release()  def main():  threads = [threading.Thread(target=increase\_by\_one), threading.Thread(target=increase\_by\_one)]  for thread in threads:  thread.start()   for thread in threads:  thread.join()   print(value) # should be 200000  if \_\_name\_\_ == "\_\_main\_\_":  main() |
| --- |

Przykład: Użycie lock wraz z Context Managarem (zamiast acquire oraz release).

| def increase\_by\_one():  global value   with lock:  for i in range(100000):  value += 1 |
| --- |

Wynik:



W powyższym przykładzie widzisz, jak I wątek zablokował dostęp do zasobów do momentu, kiedy sam nie przestał przetwarzać danych w funkcji. Z tego powodu wątek II nie mógł przeszkodzić pierwszemu w wykonywaniu operacji. Tak więc proces inkrementacji zmiennej value nie mógł zostać zaburzony i ostatecznie otrzymaliśmy oczekiwaną wartość równą 200000.

1. **DeadLocks**

Zanim przejdziemy dalej, musimy zatrzymać się przy potencjalnym problemie mogącym pojawiać wraz z lock-ami. Jest to tzw. deadlock, który pojawia się w momencie, gdy oba wątki blokują siebie nawzajem lub gdy próbujemy zablokować wątek, który już wcześniej wywołał metodę .acquire() oraz jej nie uwolnił przez .release().

**Przykład:** *Deadlock*

| import threading  def do\_deadlock():  lock = threading.Lock()  print("Before first acquire")  lock.acquire()  print("Before second acquire")  lock.acquire()  lock.release()  print("You'll never come over here!")  def main():  do\_deadlock()  if \_\_name\_\_ == "\_\_main\_\_":  main() |
| --- |

**Zadanie:**

Spróbuj uruchomić powyższy program. Jaki będzie efekt?

**Wyjaśnienie:**

Problem w powyższym przykładzie wynika z faktu, iż dwukrotnie alokujemy lock-a, uprzednio go nie zwalniając. Tak więc w momencie wywołania linii lock.acquire() po raz drugi, wątek zaczyna czekać na zwolnienie uprzednio zalokowanego lock-a.

Problemem jest fakt, że aby lock ten został zwolniony, musimy dojść do linii lock.release(), która nigdy nie zostanie osiągnięta.

Deadlocki więc pojawiają się najczęściej w sytuacjach, gdy:

1. Niewłaściwie zwalniamy utworzone locki
2. Nie zadbaliśmy o poprawny design programu i, np. jeden wątek czeka w nieskończoność na zwolnienie lock-a przez inny wątek.

Deadlock możesz więc rozumieć jak sytuację, w której wątek A czeka aż wątek B skończy swoją operację a wątek B czeka aż wątek A zakończy akcję

**Dygresja:**

Sytuacją odwrotną do deadlock jest zjawisko **livelock-a**. Występuje on, gdy dwa procesy, aby uniknąć deadlock-a zatrzymują wykonywanie kodu, aby dać możliwość innym wątkom na wykonanie się. Aby ułatwić zrozumienie, wyobraź sobie sytuację, gdy dwie osoby na korytarzu, aby minąć siebie, wybierają ciągle tą samą trasę, kończąc na ciągłej wzajemnej blokadzie.

1. **Program wieloprocesowy (moduł multiprocessing)**

W sekcji dotyczącej GILa wspomniałem o tym, że chcąc faktycznie uruchamiać programy równolegle i przyspieszać wykonywanie pewnych procedur wykorzystujących duży narzut obliczeniowy procesora, powinniśmy tworzyć kilka równoległych procesów w obrębie jednego programu.

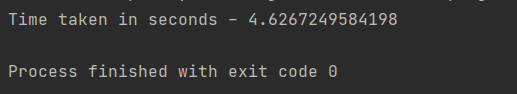
Istnieje wiele sposobów na utworzenie środowiska składającego się z kilku procesów, jednak najbardziej podstawowym jest wykorzystanie, tzw. **pool** w **asynchronicznym** (równoległym) podejściu.

Dla lepszego zobrazowania, przedstawię Ci i porównaj wydajność programu napisanego w sposób wielowątkowy jak i przy wykorzystaniu kilku procesów.

**Przykład:** *Program wielowątkowy do odliczania czasu.*

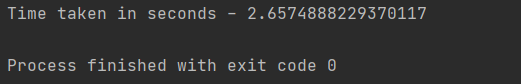
| import time from threading import Thread  TIME = 25000000  def countdown(n: int):  while n > 0:  n -= 1  if \_\_name\_\_ == "\_\_main\_\_":  t1 = Thread(target=countdown, args=[TIME])  t2 = Thread(target=countdown, args=[TIME])   start = time.time()  t1.start()  t2.start()  t1.join()  t2.join()  end = time.time()   print('Time taken in seconds -', end - start) |
| --- |

**Wynik:**



**Przykład:** *To samo, ale na równolegle wykonujących się procesach.*

| from multiprocessing import Pool import time  TIME = 25000000  def countdown(n: int):  while n > 0:  n -= 1  if \_\_name\_\_ == '\_\_main\_\_':  pool = Pool(processes=2)  start = time.time()  p1 = pool.apply\_async(countdown, [TIME])  p2 = pool.apply\_async(countdown, [TIME])  pool.close()  pool.join()  end = time.time()  print('Time taken in seconds -', end - start) |
| --- |



**Wyjaśnienie:**

Zwróć uwagę, jak bardzo poprawiliśmy czas wykonania programu, zrównoleglając wykonanie operacji na kilku procesach!

1. Aby móc stworzyć, tzw. pool konieczny był import biblioteki **multiprocessing:**

from multiprocessing import Pool

1. Obiekt Pool stworzyliśmy wraz z parametrem określającym, z ilu ma się on składać procesów. Pool umożliwia utworzenie kilku procesów i rozdzielenie między nimi obliczeń procesora między jego kilkoma rdzeniami.
2. Linie:

p1 = pool.apply\_async(countdown, [TIME])  
p2 = pool.apply\_async(countdown, [TIME])

Umożliwiają uruchomienie kilku procesów asynchronicznie.

1. pool.close() zapobiega przed wprowadzeniem dodatkowych workerów (procesów) do poola

pool.join() tak jak przy wątkach, dzięki dołączeniu procesów do procesu głównego programu, następuje oczekiwanie na zakończenie działania wszystkich procesów; w ten sposób wystrzegamy się przed przedwczesnym zakończeniem działania programu (gdy procesy jeszcze nie zakończyły swojej pracy)