

Kurs języka Python

Wykład 9.

Marcin Młotkowski

14 grudnia 2009

- 1 Wątki
- 2 Współdzielone zasoby wątków

Wstęp

Z Wikipedii:

Wątek (ang. thread) — to jednostka wykonawcza w obrębie jednego procesu, będąca kolejnym ciągiem instrukcji wykonywanym w obrębie tych samych danych (w tej samej przestrzeni adresowej).

Wątki tego samego procesu korzystają ze wspólnego kodu i danych, mają jednak oddzielne stosy.

Po co używać wątków

- zrównoleglenie wolnych operacji wejścia/wyjścia (ściągnięcie pliku/obsługa interfejsu)
- jednoczesna obsługa wielu operacji, np. serwery WWW

Przykładowe obliczenie programu dwuwątkowego

Przykład 1.

Wątek I	Wątek II
<code>i = i + 1</code>	
<code>print i</code>	
<code>i = i + 1</code>	
<code>print i</code>	
<code>i = i + 1</code>	
<code>print i</code>	
	<code>i = i + 1</code>
	<code>print i</code>
	<code>i = i + 1</code>
	<code>print i</code>
	<code>i = i + 1</code>
	<code>print i</code>

Przykład 2.

Wątek I	Wątek II
<code>i = i + 1</code>	
	<code>i = i + 1</code>
<code>print i</code>	
	<code>print i</code>
<code>i = i + 1</code>	
	<code>i = i + 1</code>
<code>print i</code>	
	<code>print i</code>
<code>i = i + 1</code>	
	<code>i = i + 1</code>
<code>print i</code>	
	<code>print i</code>

Wątki w Pythonie

- `thread` (3.0: `_thread`): niskopoziomowa biblioteka
- `threading`: wysokopoziomowa biblioteka
- `dummy_thread`
- `dummy_threading`
- `multiprocessing`

Wątki w Pythonie

- `thread` (3.0: `_thread`): niskopoziomowa biblioteka
- `threading`: wysokopoziomowa biblioteka
- `dummy_thread`
- `dummy_threading`
- `multiprocessing`

Jak korzystać z wątków

```
moduł threading
```

```
class Thread:
```

```
    def run(self):
```

```
        """Operacje wykonywane w wątku"""
```

```
    def start(self):
```

```
        """Wystartowanie obliczeń w wątku"""
```

Przykładowe zadanie

Zasymulowanie za pomocą wątków biegaczy w maratonie.

Implementacja klasy biegaczy

```
import threading

total_distance = 0

class runner(threading.Thread):
    def __init__(self, nr_startowy):
        self.numer = nr_startowy
        threading.Thread.__init__(self)
```

Implementacja biegu

```
class runner, cd
def run(self):
    global total_distance
    dystans = 42195
    while dystans > 0:
        dystans = dystans - 1
        total_distance = total_distance + 1
        if dystans % 10000 == 0:
            print 'Zawodnik nr %i' % self.numer
    print 'Zawodnik %i na mecie' % self.numer
```

Rozpoczęcie biegu

```
r1 = runner(1)
r2 = runner(2)

r1.start()
r2.start()

r1.join()
r2.join()

print 'koniec wyścigu, dystans', total_distance
```

Rola `.join`

- Główny program to też wątek, więc po wywołaniu

`r1.start()`

są dwa wątki

- `r1.join()` oznacza, że wątek nadrzędny będzie czekał na zakończenie wątku `r1`

Dostęp do wspólnej zmiennej wątków

Przypomnienie

```
total_distance = 0
```

```
class runner(threading.Thread):
```

```
    ...
```

```
        total_distance = total_distance + 1
```

```
print total_distance
```

Zagadka

Jaka jest wartość zmiennej `total_distance`?

Teoria

$2 * 42195 = 84390$

Praktyka

54390

74390

83464

...

Zagadka

Jaka jest wartość zmiennej `total_distance`?

Teoria

$2 * 42195 = 84390$

Praktyka

54390

74390

83464

...

Zagadka

Jaka jest wartość zmiennej `total_distance`?

Teoria

$2 * 42195 = 84390$

Praktyka

54390

74390

83464

...

Operacje atomowe?

```
i = i + 1
```

```
LOADFAST 0
```

```
LOAD_CONST 1
```

```
BINARY_ADD
```

```
STORE_FAST 0
```

Operacje atomowe?

```
i = i + 1
```

```
LOADFAST 0  
LOAD_CONST 1  
BINARY_ADD  
STORE_FAST 0
```

```
i = i + 1
```

```
LOADFAST 0  
LOAD_CONST 1  
BINARY_ADD  
STORE_FAST 0
```

Blokady

Klasa Lock

```
lock = Lock()
```

```
def run(self):
```

```
    global lock
```

```
    ...
```

```
        lock.acquire()
```

```
        total_distance = total_distance + 1
```

```
        lock.release()
```

Inne blokady

RLock

Wątek może założyć blokadę dowolną liczbę razy, i tyleż razy musi ją zwolnić. Bardzo spowalnia program.

Semaphore

Blokadę można założyć ustaloną liczbę razy:

```
sem = Semaphore(3)
sem.acquire()
sem.acquire()
sem.acquire()
sem.acquire() # blokada
```

Czekanie na zasób

Jeden wątek (barman) nalewa mleko do szklanki, drugi (klient) czeka na napełnienie szklanki do pełna i wypija mleko.

Implementacja picia mleka

```
lck = Lock()
```

Nalewanie

```
lck.acquire()  
for i in range(5):  
    szklanka_mleka = szklanka_mleka + 1  
lck.release()
```

Wypijanie

```
while szklanka_mleka != 5: pass  
lck.acquire()  
while szklanka_mleka > 0:  
    szklanka_mleka = szklanka_mleka - 1  
lck.release()
```

Implementacja picia mleka

```
lck = Lock()
```

Nalewanie

```
lck.acquire()  
for i in range(5):  
    szklanka_mleka = szklanka_mleka + 1  
lck.release()
```

Wypijanie

```
while szklanka_mleka != 5: pass  
lck.acquire()  
while szklanka_mleka > 0:  
    szklanka_mleka = szklanka_mleka - 1  
lck.release()
```

Zmienne warunkowe

Mechanizm który pozwala na usypianie i budzenie wątków.

Implementacja

```
lck = threading.Condition()
```

Konsumpcja

```
lck.acquire()
```

```
while szklanka_mleka != 5:
```

```
    lck.wait()
```

```
while szklanka_mleka > 0: szklanka_mleka = szklanka_mleka - 1
```

```
lck.release()
```

Nalewanie

```
lck.acquire()
```

```
for i in range(5):
```

```
    szklanka_mleka = szklanka_mleka + 1
```

```
lck.notify()
```

```
lck.release()
```

Zmienne warunkowe

- Zmienne warunkowe są zmiennymi działającymi jak blokady (`acquire()`, `release()`);
- metoda `wait()` zwalnia blokadę i usypia bieżący wątek;
- metoda `notify()` budzi jeden z uspiomych wątków (na tej zmiennej warunkowej), `notifyAll()` budzi wszystkie uspiome wątki.

Wady takiego mechanizmu

- jest tylko jedna szklanka, można do niej tylko nalewać albo tylko z niej pić;
- barman nie może nalać więcej szklanek na zapas i iść do domu

Struktury danych do programów wielowątkowych

Klasa Queue:

- Jest to kolejka FIFO, thread-safe;
- Konstruktor: `Queue(rozmiar)`
- pobranie elementu (z usunięciem): `.get()`; gdy kolejka jest pusta zgłasza wyjątek `Empty`
- `.get(True)`: gdy kolejka jest pusta, wątek jest usypiany;
- umieszczenie elementu: `.put(element)`, gdy kolejka jest pełna to zgłaszany jest wyjątek `Full`;
- umieszczenie elementu: `.put(element, True)`, gdy kolejka jest pełna wątek jest usypiany;
- `.full()`, `.empty()`

Warianty klasy Queue

- LifoQueue
- PriorityQueue