Akademia Handlowa Nauk Stosowanych w Radomiu

Programowanie równoległe
Laboratorium 4

Radom 2022/2023

1. Cel zadania

Obecnie niesłabnącym zainteresowaniem cieszą się języki interpretowane. Ze względów formalnych oraz potrzeb wynikających bezpośrednio z rozwiązań nowoczesnego programowania warto posiadać wiedzę dotyczącą tworzenia oraz obsługi wątków w językach JavaScript oraz Python, które są obecnie najpopularniejszymi językami skryptowymi.

2. Potrzebne narzędzia.

- kompletny zestaw komputerowy z co najmniej dwurdzeniowym procesorem
- dowolny system operacyjny Windows, Linux, macOS bądź równoważny
- dowolna przeglądarka lub node.js (JavaScript), środowisko Python
- dowolne narzędzie programistyczne (np. Microsoft Visual Code, PyCharm i podobne)

3. Informacje wstępne

Wielowątkowość to obecnie standard w programowaniu. Dzięki wielowątkowości aplikacje stają się co najmniej bardziej responsywne - poszczególne zadania mogą wykonywać się niezależnie od wątku, w którym następuje obsługa zdarzeń generowanych przez użytkownika (ruch myszą, zdarzenia klawiatury i inne).

a) Aplikacje HTML same w sobie zajmują dokładnie jeden wątek - generowanie strony WWW. Jeżeli jednak strona będzie podczas ładowania i późniejszego użytkowania wykonywała dodatkowe operacje, przykładowo będzie ładowała dynamicznie dane z baz danych lub rozpocznie renderowanie grafiki w zależności od pozycji kursora użytkownika, to użytkownik może odczuwać dyskomfort przy przeglądaniu naszej witryny – dozna charakterystycznego zacinania się dokumentu, być może nawet całego okna przeglądarki (a przynajmniej zakładki, na której wyświetlona zostanie nasza witryna).

Chociaż JavaScript nie operuje bezpośrednio na wątkach użytkownika, wprowadzono w niej mechanizm tak zwanych głębokich wątków - programista wskazuje operacje, które powinny wykonać się w osobnym wątku, a sam wątek tworzy i obsługuje (w roli pośrednika) środowisko uruchomieniowe.

Domyślnie JavaScript należy do rodziny języków asynchronicznych zdarzeniowych. Oznacza to, że wykonuje ona określone operacje w reakcji na określone zdarzenia, które nie mają określonego czasu wystąpienia - są losowe, jak przykładowo kliknięcia myszy użytkownika lub jako odpowiedzi nadsyłane z serwera bądź z zasobów lokalnych. Rozwiązanie to rozwiązuje blokowe wykonywanie skryptu w sekcjach krytycznych algorytmu. Przykładowo ładowanie strony WWW nie przerywa się podczas wczytywania kolejnych danych z serwera – gotowa część może od razu zostać przedstawiona użytkownikowi, a gdy skrypt otrzyma pozostałe dane JavaScript, poprzez odpowiednie odwołania do elementów strony, może je efektywnie załadować w sposób niewidoczny dla użytkownika.

Powyższe rozwiązanie przestało w pełni wystarczać w chwili, gdy strony WWW stały się pełnoprawnymi aplikacjami, w których dokument HTML stanowi jedynie graficzny interfejs aplikacji ulokowanej na odległym serwerze. Wczytywanie danych do tabel, odświeżanie wyświetlanych danych w reakcji na działania innych użytkowników czy operacje, które zleca sam użytkownik, wymagają dodatkowych mechanizmów podziału zadań przez skrypt. Do tego służą elementy zwane Promise – odpowiedniki wątków w innych językach.

Najprostsza postać Promise prezentuje się następująco:

```
let newPromise = new Promise(function(resolve, reject) {
  resolve('Sukces');
  reject('Porażka');
```

```
});

nwePromise.then(
  function(value) { /* kod sukcesu */ },
  function(error) { /* kod porażki */ }
);
```

Pierwsza część kodu tworzy nowy "wątek" przyjmujący dwa parametry - funkcję, która wykona się w przypadku w pełni spełnionej obietnicy (kod nie napotka błędów i zostanie wykonany) lub kod napotka błędy i zostanie wykonany alternatywny kod zawarty pod funkcją reject.

W drugiej części części wywołujemy wcześniej utworzony wątek. Pod wcześniej wskazane funkcje sukcesu i porażki podstawiamy kod, który ma zostać wykonany; należy mieć na uwadze, że parametry w każdej z funkcji są wartościami zwracanymi przez kod Promise (w naszym wypadku odpowiednio ciągi znakowe Sukces oraz Porażka).

Trzeba pamiętać, że to, co zwróci Promise zależy tylko od nas (sami możemy stworzyć odpowiedni kod warunkowy, który doprowadzi do zwrócenia odpowiedniej funkcji).

Prócz powyższego mechanizmu JavaScript oferuje również możliwość tworzenia asynchronicznych funkcji oraz blokowania domyślnie asynchronicznych elementów językowych. Przykładowo funkcja:

```
async function myFunction() {
  return "Hello";
}
```

będzie zachowywała się tak samo jak element Promise. Jej wywołanie będzie miało następującą postać:

```
myFunction().then(
  function(value) { /* kod do wykonania */ }
```

Wartość value w powyższym przykładzie przyjmie wartość Hello.

Prócz słowa async JavaScript posiada także słowo kluczowe await. Słowo to może być używane jedynie w połączeniu w elementem asynchronicznym (wewnątrz niego) i daje nam gwarancję, że kod nie wykona się dalej do chwili, w której element poprzedzony await nie zakończy się. Przykładem takiego rozwiązania może być poniższy kod:

```
async function wyswietl() {
  let watek = new Promise(function(resolve) {
    resolve("Witaj !!");
  });
  document.getElementById("demo").innerHTML = await watek;
}

wyswietl ();
```

Warto mieć na uwadze, że dzięki niektórym modyfikacjom silnika V8 rozwiązanie async/await bywa szybsze niż stosowanie pojedynczych elementów Promise.

b) Python staje się coraz popularniejszym językiem, głównie za sprawą swojej prostoty, która czyni go coraz ważniejszym narzędziem do pracy ze sztuczną inteligencją. Język, którego podstawowym zadaniem było tworzenie skryptów automatyzujących testowanie innych aplikacji, stał

się podstawowym narzędziem do tworzenia i rozwijania aplikacji w szczególności bazujących na przetwarzaniu dużej ilości informacji – big data, modele językowe, ogólna AI. Ponadto społeczność skupiona wokół tego języka stworzyła wiele dodatków, które umożliwiły tworzenie w tym języku dowolnych aplikacji i rozwiązań informatycznych – od prostych aplikacji, poprzez aplikacje do zarządzania i modyfikowania ustawień systemowych, po aplikacje webowe. Python doczekał się również implementacji na niektórych urządzeniach wbudowanych – w szczególności do obsługi inteligentnych domów.
Wszystko to powoduje, że programiści używający tego języka również potrzebują i wymagają większej elastyczności tworzonych rozwiązań. Dlatego też Python posiada odpowiednie moduły odpowiadające za wielowątkowość/wieloprocesowość: treading oraz multiprocessing.

Przykład kodu z wykorzystaniem wątkowości:

```
def watek(zmienne):
print('Jestem wątkiem', zmienne)

for i in range(10):
w = threading.Thread(target=watek, args=(i,))
w.start()
```

Utworzone zostanie 10 wątków, które jako parametr przyjmą wartość zmiennej i.

```
def watek(zmienne):
print('Jestem wątkiem', zmienne)
time.sleep(random.randrange(1,100))
print('Watek',zmienne,'kończy działanie')

print('Start właściwego programu')
for i in range(10):
w = threading.Thread(target=watek, args=(i,))
w.start()
print('Koniec właściwego programu')
```

Zmodyfikowany przykład pokazujący niezależność wykonywanych wątków od głównej części kodu.

```
def watek(zmienne):
print('Jestem wątkiem', zmienne)
time.sleep(random.randrange(1,100))
print('Watek',zmienne,'kończy działanie')

watki = list()
print('Start właściwego programu')
for i in range(10):
w = threading.Thread(target=watek, args=(i,))
watki.append(w)
w.start()

for ww in watki:
ww.join()

print('Koniec właściwego programu')
```

Kod pokazujący przydatność funkcji join().

Prócz wątków Python posiada bardzo podobne rozwiązanie (z punktu widzenia programisty) zawarte w module multiprocessing:

```
def proces(zmienne):
print('Jestem procesem', zmienne)
time.sleep(random.randrange(1,100))
print('WProces',zmienne,'kończy działanie')

procesy = list()
print('Start właściwego programu')
for i in range(10):
p = multiprocessing.Thread(target=watek, args=(i,))
procesy.append(p)
p.start()

for pp in procesy:
pp.join()
```

Jak można zauważyć, kod różni się jedynie nazwą wykorzystanego modułu. Zmianie uległa też nazwa bazowej klasy (z Thread na Process). Wyniki także powinny być podobne.

UWAGA! Proszę pamiętać, że w celu poprawnego uruchomienia obu wersji przykładów w kodzie koniecznie trzeba dodać odpowiedni import (import threading lub import multiprocessing). Bez wskazanych importów wskazany kod może się nie uruchomić.

4. Przebieg.

Laboratorium ma przebieg teoretyczno-badawczy. Głównym motywem sporządzanego sprawozdania powinna być równoległość w językach interpretowanych. W ramach sprawozdania należy pochylić się na takich aspektach jak:

- czym jest GIL (Global Interpreter Lock); do czego służy, jakie ma zadania, jakie są wady i zalety jego zastosowania
- jaka jest realna różnica, w przypadku JavaScript, pomiędzy wykonywaniem czystego Promise a Promise wykorzystującym mechanizmy async/await
- czym różni się model Promise od Workers w JavaScript? Jakie są zastosowania jednego i drugiego rozwiązania?
- w jaki sposób w języku Python funkcjonują wątki, a w jaki procesy?
- w jaki sposób można nawiązywać współpracę pomiędzy wątkami w przedstawionych językach? Jakie wbudowane mechanizmy/metody do współpracy/kooperacji posiada każdy z języków?
- Czy efektywność wielowątkowości języków intepretowanych może być zadawalająca dla algorytmów obliczeniowych? Można oprzeć się wynikami i konkluzjami innych badaczy.
- Mile widziana byłaby konkluzja na temat przydatności wątków i wielowątkowości w omawianych językach.

5. Zakończenie

Rozwiązanie należy przesłać w postaci sprawozdania na adres piotr_dobosz@int.pl, w temacie wiadomości zawierając frazę [AHNS_PR].

6. Dodatki

JavaScript:
https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise
https://realpython.com/python-gil/
https://superfastpython.com/threading-vs-multiprocessing-in-python/
https://realpython.com/intro-to-python-threading/
https://www.digitalocean.com/community/tutorials/python-multiprocessing-example