Sprawozdanie 5 filozofow - semafory

November 8, 2021

1 Problem 5 filozofów rozwiązane za pomocą semaforów. Analiza różnych podejść

Paweł Kruczkiewicz

Treść:

Korzystając z wykonanych implementacji: 1. Uruchom eksperymenty dla różnej liczby filozofów i dla każdego wariantu implementacji (nie powodującego zakleszczenia). 2. Zmierz średni czas oczekiwania każdego filozofa na dostęp do widelców. Wykonaj kilka pomiarów dla każdego przypadku testowego. 3. Wyniki przedstaw na wykresach porównawczych, dbając o odpowiednią wizualizację (można wykorzystać np. wykresy pudełkowe). 4. Sformułuj i zapisz wnioski. Czy średnie czasy oczekiwania są wyższe dla wariantu z możliwością zagłodzenia? Czy brak mechanizmów syn chronizacji zwiększa czas oczekiwania na dostęp do zasobów? (Porównanie Node.js vs. Java może to być trudne do oceny z uwagi na różne środowiska wykonawcze).

[2]:
```
import os
import numpy as np
import matplotlib.pyplot as plt
```

1.1 Ad. 1

Programy uruchomiono z modyfikacją - po zakończeniu każdego oczekiwania na pałeczki, wypisywany na ekran jest komunikat z numerem filozofa oraz z czasem oczekiwania (w milisekundach). Program zatrzymuje się, gdy każdy filozof "zje" daną liczbę posiłków (w poniższym eksperymencie tą liczbą jest 20). Następnie zapisano wyjście do pliku, zgrepowano. Poniżej przedstawiono program, który parsuje dany plik wynikowy.

[12]:
```
def parse_line(line):
    # All numbers are surrounded by hash (eg. "Philosopher: #2# time: #3022# [ms]")
    splitted_line = line.split("#")
    philosopher_number, time_in_ms = int(splitted_line[1]) , int(splitted_line[3])
    return philosopher_number, time_in_ms


def filter_and_compute_avg_time(time_tuples, philosopher_number):
    filtered_values = list(map(lambda x: x[1],
```

```
                                (filter(
                                  lambda tup: tup[0] == philosopher_number,
                                  time_tuples)
                                )))
    return sum(filtered_values)/len(filtered_values)

def parse_file(filename, n=5, agr_func=filter_and_compute_avg_time):
    with open(filename, "r") as file:
        data = file.readlines()
    time_data = list(filter(lambda line: "time" in line, data))
    time_tuples = [parse_line(data_line) for data_line in time_data]
    return [agr_func(time_tuples, i) for i in range(n)]
```

Funkcja pomocnicza do tworzenia wykresów:

[15]:
```
bar_width = 0.25

class BarGroupInformation:
    def __init__(self, values, color, label, number):
        self.values = values
        self.color = color
        self.label = label
        self.range = np.arange(len(values)) + bar_width*number

def create_plot(x_labels, info_list, title):
    # Make the plot
    [plt.bar(info.range, info.values, color=info.color, label=info.label,␣
 ↪width=bar_width)
     for info in info_list]

    # Add xticks on the middle of the group bars
    plt.xlabel('Philosopher number', fontweight='bold')
    plt.xticks([r + bar_width for r in range(len(info_list[0].values))],␣
 ↪x_labels)
    plt.ylabel("Time [ms]")
    # Create legend & Show graphic

    plt.title(title)
    plt.legend()
    plt.show()
```

1.2 Ad. 2 i Ad. 3

1.2.1 Słowo wstępu

Poniżej przedstawiono wyniki eksperymentów (tj. pomiary czasów) dla niemal wszsytkich opisanych w treści zadania implementacji oprócz podejścia naiwnego w JSie - było ono jedynym, które za każdym razem powodowało zakleszczenie. Całość analizy porównawczej podzielono na 2 części wg

języka programowania, ponieważ środowiska wykonawcze dla każdego języka znacząco się różnią. ### Rozwiązanie naiwne Ponieważ poniżej poświęcono sporo czasu na rozwiązania poprawne, ten fragment zostanie poświęcony rozwiązaniu powodującemu zakleszczenie. W wyniku eksperymentu program napisany w JavaScripcie zawsze "zawieszał się" z wynikiem:

```
Philosopher 0 is thinking
Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 3 is thinking
Philosopher 4 is thinking
Philosopher 0 took the left fork
Philosopher 1 took the left fork
Philosopher 2 took the left fork
Philosopher 3 took the left fork
Philosopher 4 took the left fork
```

Dzieje się tak dlatego, że w pętli for filozofowie odkładani są na stos wywołań zawsze w tej samej kolejności, `timeout` ustawiony jest na ten sam czas, zatem wszyscy filozofowie podnoszą jeden po drugim lewy widelec, zakleszczając program.

Do trochę innej sytuacji dochodzi w javie, gdzie kazdy filozof jest osobnym wątkiem. Wtedy niedeterminizm każdego uruchomienia programu sprawia, że czasem nie mamy zakleszczenia, a czasem mamy. W większości uruchomień w trakcie przeprowadzania eksperymentu nie zauważono zakleszczenia, ale zdarzały się sytuacje takie jak przedstawiona poniżej:

```
(...)
Philosopher number 2 has lifted the left fork
Philosopher number 4 has stopped thinking
Philosopher number 3 has lifted the left fork
Philosopher number 0 has lifted the left fork
Philosopher number 1 has lifted the left fork
Philosopher number 4 has lifted the left fork
(...)
```

... co naturalnie prowadzi do zawieszenia się programu.

1.2.2 Java

W zadaniu mieliśmy przetestować 2 alogrytmy: naiwny oraz z kelnerem. Chociaż algorytm naiwny może zakleszczyć się bardzo łatwo (patrz: podpunkt wyżej), tak nie dzieje się to zawsze przy programowaniu wielowątkowym. Dlatego poniżej porównano czas obu tych podejść:

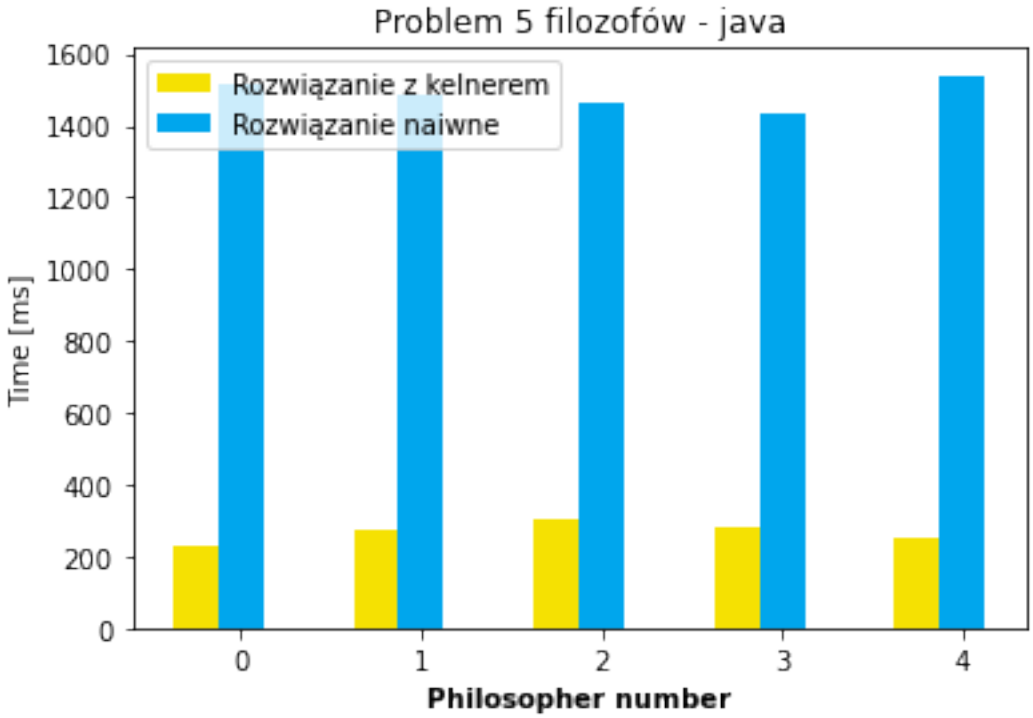
[16]:
```
directory_name = "czasy"
```

[17]:
```
# java - 5 filozofów
no_referee_time = parse_file(os.path.join(directory_name, "noReferee.txt"))
referee_time = parse_file(os.path.join(directory_name, "referee.txt"))

list_of_values = [referee_time, no_referee_time]
colors = ['#f5e102', '#00a6ed']
```

```
labels = ["Rozwiązanie z kelnerem", "Rozwiązanie naiwne"]
numbers = list(range(len(list_of_values)))
infos = [BarGroupInformation(values, color, label, number)
 for values, color, label, number in zip(list_of_values, colors, labels,␣
 ↪numbers)]

create_plot( [str(num) for num in list(range(5))], infos, "Problem 5 filozofów␣
 ↪- java")
```

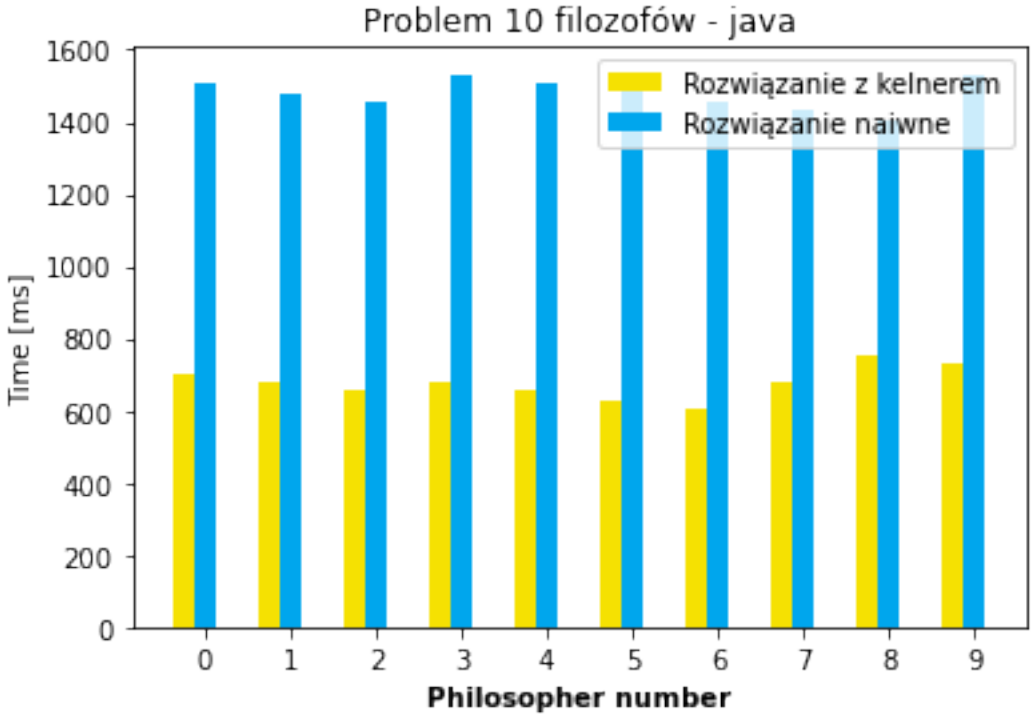


Sprawdzono również, jak zachowa się program, gdy filozofów będzie 10

[18]:
```
# java - 10 filozofów
no_referee_time_10 = parse_file(os.path.join(directory_name, "noReferee10.
 ↪txt"), 10)
referee_time_10 = parse_file(os.path.join(directory_name, "referee10.txt"), 10)

list_of_values = [referee_time_10, no_referee_time_10]
colors = ['#f5e102', '#00a6ed']
labels = ["Rozwiązanie z kelnerem", "Rozwiązanie naiwne"]
numbers = list(range(len(list_of_values)))
infos = [BarGroupInformation(values, color, label, number)
 for values, color, label, number in zip(list_of_values, colors, labels,␣
 ↪numbers)]
```

```
create_plot( [str(num) for num in list(range(10))], infos, "Problem 10␣
↪filozofów - java")
```



Jak można wywnioskować z powyższych wykresów, rozwiązanie z kelnerem, jest nie tylko bezpieczniejszym, ale i szybszym rozwiązaniem. Jednak przewaga tego rozwiązania zmniejsza się znacznie w przypadku większej liczby filozofów.

1.2.3 JavaScript

Rozwiązanie problemu 5 filozofów w JavaScripcie jest zadaniem trudniejszym ze względu na jednowątkowy charakter wykonania programów napisanych w tym języku. W implementacji, aby zapewnić asynchroniczne wykonanie programu, skorzystano z funkcji `setTimeout(func, time)`, która wykonuje daną funkcję `func` dopiero po czasie `time`.

Poniżej przedstawiono wyniki dla 3 rozwiązań problemu uctujących filozofów: 1. Rozwiązanie asymetryczne 2. Rozwiązanie z kelnerem 3. Rozwiązanie symultaniczne (czyli jednocześnie podnosimy dwa widelce tylko wtedy, gdy są wolne)

Wykres słupkowy dla 5 stołujących filozofów wygląda tak:

[19]:
```
directory_name = "czasy/js"

asym_time =  parse_file(os.path.join(directory_name, "asymetric.txt"))
conductor_time = parse_file(os.path.join(directory_name, "conductor.txt"))
```

```
simultanous_time = parse_file(os.path.join(directory_name, "simultanous.txt"))

list_of_values = [asym_time, conductor_time, simultanous_time]
colors = ['red', 'black', "green"]
labels = ["Asymetryczne", "Z kelnerem", "Symultaniczne"]
numbers = list(range(len(list_of_values)))
infos = [BarGroupInformation(values, color, label, number)
 for values, color, label, number in zip(list_of_values, colors, labels,␣
 ↪numbers)]

create_plot( [str(num) for num in list(range(5))], infos, "Problem 5 filozofów␣
 ↪- java script")
```

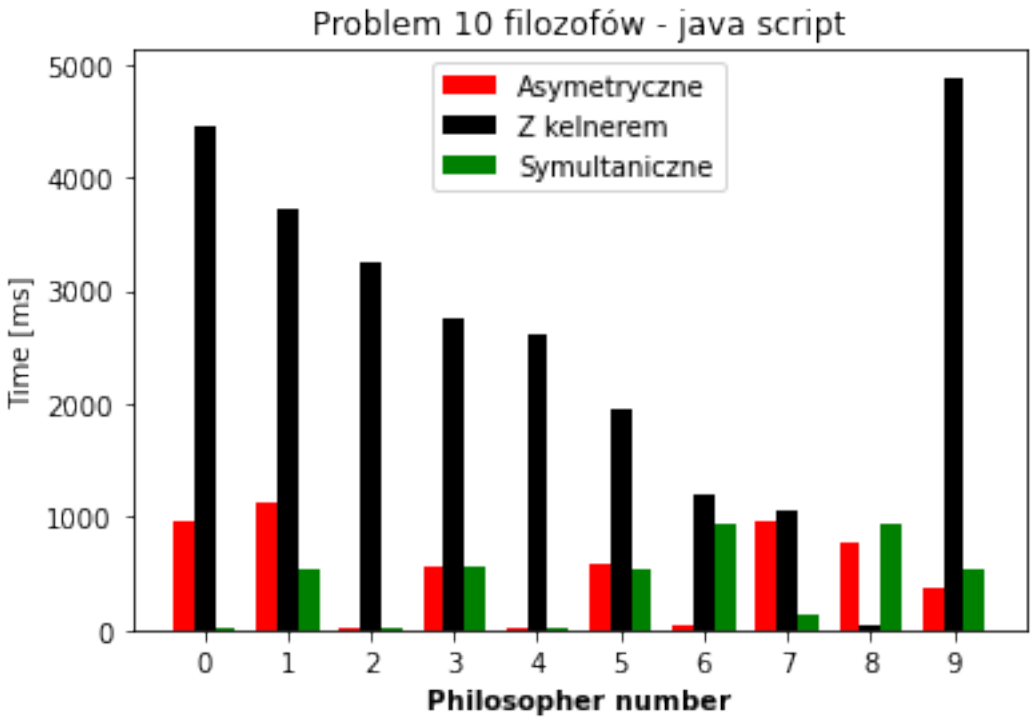
... a dla 10 tak:

[20]:
```
asym_time_10 =  parse_file(os.path.join(directory_name, "asymetric_10.txt"), 10)
conductor_time_10 = parse_file(os.path.join(directory_name, "conductor_10.
 ↪txt"), 10)
simultanous_time_10 = parse_file(os.path.join(directory_name, "simultanous_10.
 ↪txt"), 10)

list_of_values = [asym_time_10, conductor_time_10, simultanous_time_10]
colors = ['red', 'black', "green"]
```

```
labels = ["Asymetryczne", "Z kelnerem", "Symultaniczne"]
numbers = list(range(len(list_of_values)))
infos = [BarGroupInformation(values, color, label, number)
 for values, color, label, number in zip(list_of_values, colors, labels,␣
 ↪numbers)]

create_plot( [str(num) for num in list(range(10))], infos, "Problem 10␣
 ↪filozofów - java script")
```



Oba powyższe wykresy pokazują, że podejście asymetryczne oraz symultaniczne mają zdecydowaną przewagę nad rozwiązaniem z kelnerem.

Wszystkie rozwiąznia mają jednak bardzo podobną wadę: każdy z filozofów ze względu na asynchroniczne wykonanie kodu analogiczne do tego, które zaobserwowano w podejściu naiwnym (patrz dwa podpunkty wyżej) wykonywały się niemal zawsze najpierw dla jednego filozofa.

Najdrastyczniej widać to na przykładzie 10 filozofów z kelnerem. Poniżej pokazano wartości maksymalne czasu oczekiwania dla każdego filozofa:

[26]:
```
def get_max_with_given_number(tuples, number):
    return max(map(lambda x: x[1], filter(lambda x: x[0] == number, tuples)))

conductor_times_max = parse_file(os.path.join(directory_name, "conductor_10.
 ↪txt"), n=10, agr_func=get_max_with_given_number)
```

```
print("Max waiting time for each philosopher", end="\n\n")
for philosopherNum, time in enumerate(conductor_times_max):
    print(f'{philosopherNum}: {time} [ms]')
```

```
Max waiting time for each philosopher

0: 85186 [ms]
1: 62459 [ms]
2: 60831 [ms]
3: 38746 [ms]
4: 30353 [ms]
5: 28709 [ms]
6: 20036 [ms]
7: 11425 [ms]
8: 43 [ms]
9: 90111 [ms]
```

Niemal każdy filozof musiał zazwyczaj *czekać* na innego filozofa, aż ten skończy posiłek. Oznacza to, że rozwiązanie problemu 5 filozofów zaimplementowane tak, jak w tym zadaniu, byłoby bardzo złym rozwiązaniem w przypadku nieskończonej liczby posiłków - czasy oczekiwania dla niektórych filozofów zwiększałyby się do nieskończoności, czyli mielibyśmy do czynienia z zagłodzeniem. Jedyny sposób, w jaki można by było to poprawić, to wprowadzić różne czasy oczekiwania na jedzenie i myślenie, co mogłoby doprowadzić do zróżnicowania stosu wywołań interpretatora JSa.

Warto też zauważyć, że dla każdego rozwiązania można dla różnej liczby filozofów zobaczyć kilka ciekawych zależności, które wiążą się bezpośrednio z implementacją zadania oraz interpretera JSa: - w asymetrycznym zawsze drugi i czwarty filozof je najszybciej; proporcje między filozofami o numerach od 0 do 4 są z gubsza takie same dla 5 i 10 filozofów; - podobną rzecz można zauważyć dla symultanicznego, lecz tam najszybsi są 0 i 2 filozof; - w rozwiązaniu z kelnerem najszybszy jest filozof przedostatni, potem przedprzedostatni, itd. aż do ostatniego; czas zwiększa się z grubsza liniowo z każdym filozofem w tej sekwencji;

Wyniki z pewnością byłyby bardziej przewidywalne, jeżeli nie użyto by algorytmu BEB. Podobnego determinizmu próżno szukać w wielowątkowych programach.

1.3 Ad. 4

Wnioski

W niniejszym sprawozdaniu sprawdzono podejście wielowątkowe i asynchroniczne do problemu 5 filozofów na przykładzie języków Java i JavaScript. Podsumowując przedstawione wyżej opisy, można napisać, że: - warianty z możliwością zagłodzenia mają zdecydowanie wyższe czasy oczekiwania (przykładem jest tutaj kelner zaimplemenrowany w JavaScripcie) - więcej mechanizmów synchronizacji nie zawsze oznacza dłuższy czas wykonania programu (co pokazano na przykładzie implementacji w języku Java) - asynchroniczne rozwiązania zapewniają wiekszą kontrolę nad pisanym kodem niz wielowątkowe (są bardziej deterministyczne - w rozwiązaniach asynchronicznych jedynym “losowym” elementem jest algorytm BEB oraz wywoływanie funkcji, w wielowątkowych nie można przewidzieć w pełni, jaki przeplot funkcji atomicznych może się zdarzyć)

[ ]: