Teoria Współbieżności
Sprawozdanie z ćwiczenia 1 - Problem pięciu filozofów

Maciej Wiśniewski
Grupa 9 środa 8.00

1. Opis zadania

Problem pięciu filozofów jest jednym z klasycznych problemów teorii współbieżności. Podstawowe sformułowanie problemu jest następujące :

- N filozofów zasiada przy okrągłym stole
- Pomiędzy sąsiednimi filozofami leży widelec (łącznie jest N widelców)
- Każdy filozof działa ciągle według schematu „myślenie - jedzenie - myślenie - jedzenie - ...". Każdy z etapów (myślenie i jedzenie) jest skończony.
- Aby zjeść, filozof musi podnieść oba sąsiadujące widelce

Celem zadania było zaprojektowanie algorytmu jednoczesnej alokacji współdzielonych zasobów (widelce) przez konkurujące procesy (filozofowie), tak aby uniknąć zakleszczenia i zagłodzenia.

Problem został zaimplementowany w języku Java na sześć następujących sposobów:

- rozwiązanie naiwne
- rozwiązanie z możliwością zagłodzenia
- rozwiązanie asymetryczne
- rozwiązanie stochastyczne
- rozwiązanie z arbitrem
- rozwiązanie z jadalnią

2. Opis rozwiązań
   I. Rozwiązanie naiwne (z możliwością blokady)

Każdy filozof czeka, aż wolny będzie lewy widelec, a następnie go podnosi (zajmuje), następnie podobnie postępuje z prawym widelcem.

```
for (int i = 0; i < meals; i++) {
    think();

    synchronized (leftFork) {
        System.out.println("Filozof " + id + " podnosi lewy widelec " + leftFork.getId());

        synchronized (rightFork) {
            System.out.println("Filozof " + id + " podnosi prawy widelec " + rightFork.getId());
            eat();
            System.out.println("Filozof " + id + " odkłada prawy widelec " + rightFork.getId());
        }

        System.out.println("Filozof " + id + " odkłada lewy widelec " + leftFork.getId());
    }
}
```

W rozwiązaniu użyto **synchronized**. Każdy filozof na początku synchroniczne próbuje podnieść lewy widelec, a jeśli mu się to uda to podnosi prawy. Podczas działania może wystąpić deadlock - gdy wszyscy filozofowie jednocześnie podniosą lewy widelec i będą czekać na prawy.

II. Rozwiązanie z możliwością zagłodzenia

Każdy filozof sprawdza czy oba sąsiednie widelce są wolne i dopiero wtedy zajmuje je jednocześnie. Rozwiązanie to jest wolne od blokady, jednak w przypadku, gdy zawsze któryś z sąsiadów będzie zajęty jedzeniem, nastąpi zagłodzenie, gdyż oba widelce nigdy nie będą wolne.

```
for (int i = 0; i < meals; i++) {
    think();

    while (true) {
        if (leftFork.tryAcquire()) {
            if (rightFork.tryAcquire()) {
                System.out.println("Filozof " + id + " podnosi oba widelce");
                eat();
                rightFork.release();
                leftFork.release();
                System.out.println("Filozof " + id + " odkłada oba widelce");
                break;
            } else {
                leftFork.release();
            }
        }
        Thread.sleep(10);
    }
}
```

Takie rozwiązanie unika **deadlocka**, bo filozof może zrezygnować i spróbować ponownie. Tutaj rozwiązanie jest podobne do naiwnego, ale jeśli tryAcquire() na prawym widelce się nie uda to filozof zwalnia lewy widelec. Niestety możliwe jest tutaj **zagłodzenie:** niektórzy filozofowie mogą mieć mniej szczęścia i nigdy nie zdobyć obu widelców. Implementacja **ReentrantLock** z wariantem **tryLock**.

III. Rozwiązanie asymetryczne

Filozofowie są ponumerowani. Filozof z parzystym numerem najpierw podnosi prawy widelec, filozof z nieparzystym numerem najpierw podnosi lewy widelec.

```
for (int i = 0; i < meals; i++) {
    think();
    if (id % 2 == 0) {
        rightFork.acquire();
        System.out.println("Filozof " + id + " (parzysty) podnosi prawy widelec " + rightFork.getId());
        leftFork.acquire();
        System.out.println("Filozof " + id + " (parzysty) podnosi lewy widelec " + leftFork.getId());
    } else {
        leftFork.acquire();
        System.out.println("Filozof " + id + " (nieparzysty) podnosi lewy widelec " + leftFork.getId());
        rightFork.acquire();
        System.out.println("Filozof " + id + " (nieparzysty) podnosi prawy widelec " + rightFork.getId());
    }

    eat();

    leftFork.release();
    rightFork.release();
    System.out.println("Filozof " + id + " odkłada widelce");
}
```

Jest to proste rozwiązanie, które eliminuje możliwość występowania deadlocka. Zostało zaimplementowane przy użyciu **ReentrantLock** w wariancie klasycznym. Tutaj filozof bezpośrednio podnosi(acquire()) jeden widelec i od razu potem drugi.

IV. Rozwiązanie stochastyczne

Każdy filozof rzuca monetą tuż przed podniesieniem widelców i w ten sposób decyduje, który najpierw podnieść - lewy czy prawy.

```
for (int i = 0; i < meals; i++) {
    think();

    boolean leftFirst = random.nextBoolean();
    Fork first = leftFirst ? leftFork : rightFork;
    Fork second = leftFirst ? rightFork : leftFork;

    System.out.println("Filozof " + id + " rzuca monetą: " + (leftFirst ? "LEWY pierwszy" : "PRAWY pierwszy"));

    while (true) {
        if (first.tryAcquire()) {
            System.out.println("Filozof " + id + " podnosi pierwszy widelec " + first.getId());
            if (second.tryAcquire()) {
                System.out.println("Filozof " + id + " podnosi drugi widelec " + second.getId());
                eat();
                second.release();
                first.release();
                System.out.println("Filozof " + id + " odkłada widelce");
                break;
            } else {
                first.release();
            }
        }
        Thread.sleep(10);
    }
}
```

Rozwiązania zostały zaimplementowane z użyciem Random i **AtomicBoolean**. Po wylosowaniu kolejności filozof próbuje podnieść najpierw jeden potem drugi, jeśli podniesie pierwszy, ale nie uda mu się podnieść drugiego filozof zwalnia pierwszy.

V. Rozwiązanie z arbitrem

Zewnętrzny arbiter (lokaj, kelner) pilnuje, aby jednocześnie co najwyżej czterech (w ogólnym przypadku N-1) filozofów konkurowało o widelce. Każdy podnosi najpierw lewy, a potem prawy widelec. Jeśli naraz wszyscy filozofowie będą chcieli jeść, arbiter powstrzymuje jednego z nich aż do czasu, gdy któryś z filozofów skończy jeść.

```
for (int i = 0; i < meals; i++) {
    think();

    arbiter.acquire();
    System.out.println("Filozof " + id + " otrzymał pozwolenie od arbitra");

    synchronized (leftFork) {
        System.out.println("Filozof " + id + " podnosi lewy widelec " + leftFork.getId());

        synchronized (rightFork) {
            System.out.println("Filozof " + id + " podnosi prawy widelec " + rightFork.getId());
            eat();
            System.out.println("Filozof " + id + " odkłada widelce");
        }
    }

    arbiter.release();
    System.out.println("Filozof " + id + " zwraca pozwolenie arbitrowi");
}
```

Rozwiązanie zapobiega deadlockowi i zapewnia pewną kontrolę sprawiedliwości. Implementacja jest oparta o **Semafora**. Arbiter czuwa nad dostępnością(acquire()). Filozof po otrzymaniu pozwolenia najpierw synchronicznie podnosi lewy potem prawy widelec.

VI. Rozwiązanie z jadalnią

Rozwiązanie jest modyfikacją wersji z arbitrem. Filozof, który nie zmieści się w jadalni (czyli arbiter nie pozwolił mu jeść) je „na korytarzu” podnosząc jednorazowo widelce w odwrotnej kolejności (do reszty filozofów w jadalni).

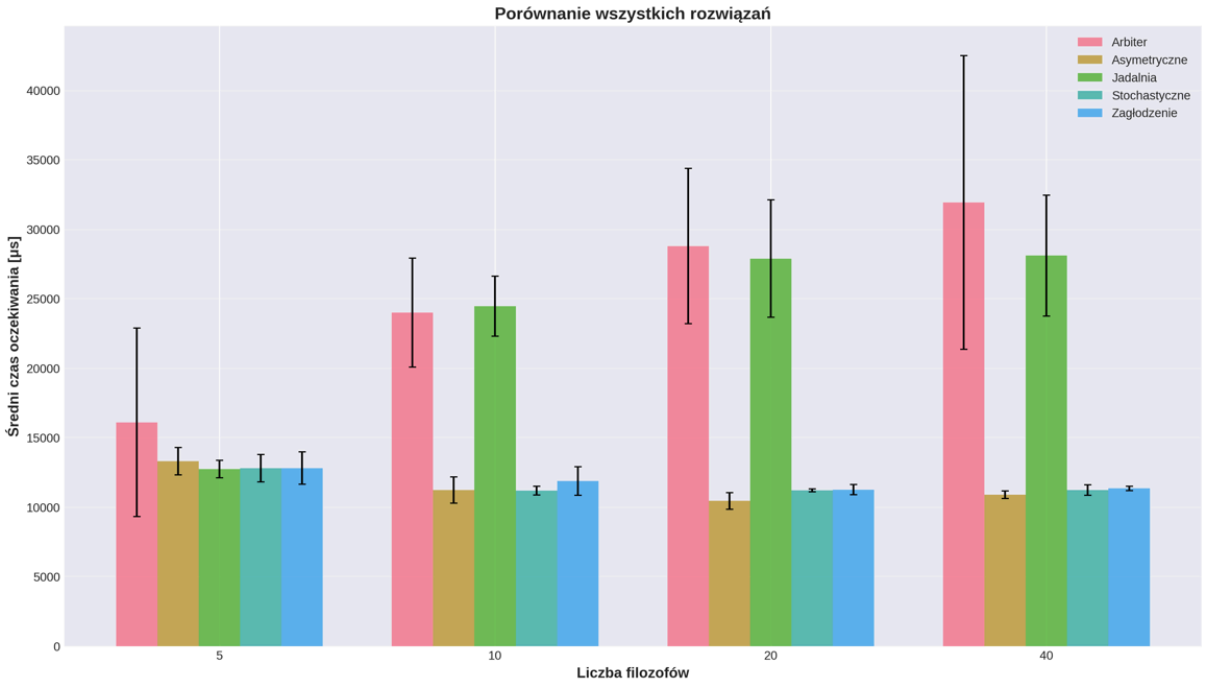
```
for (int i = 0; i < meals; i++) {
    think();

    if (diningRoom.tryAcquire()) {
        System.out.println("Filozof " + id + " wchodzi do jadalni");

        synchronized (leftFork) {
            System.out.println("Filozof " + id + " (w jadalni) podnosi lewy widelec " + leftFork.getId());

            synchronized (rightFork) {
                System.out.println("Filozof " + id + " (w jadalni) podnosi prawy widelec " + rightFork.getId());
                eat();
                System.out.println("Filozof " + id + " (w jadalni) odkłada widelce");
            }
        }

        diningRoom.release();
        System.out.println("Filozof " + id + " opuszcza jadalnię");
    } else {
        System.out.println("Filozof " + id + " je na korytarzu (odwrotna kolejność)");

        synchronized (rightFork) {
            System.out.println("Filozof " + id + " (korytarz) podnosi prawy widelec " + rightFork.getId());

            synchronized (leftFork) {
                System.out.println("Filozof " + id + " (korytarz) podnosi lewy widelec " + leftFork.getId());
                eat();
                System.out.println("Filozof " + id + " (korytarz) odkłada widelce");
            }
        }
    }
```

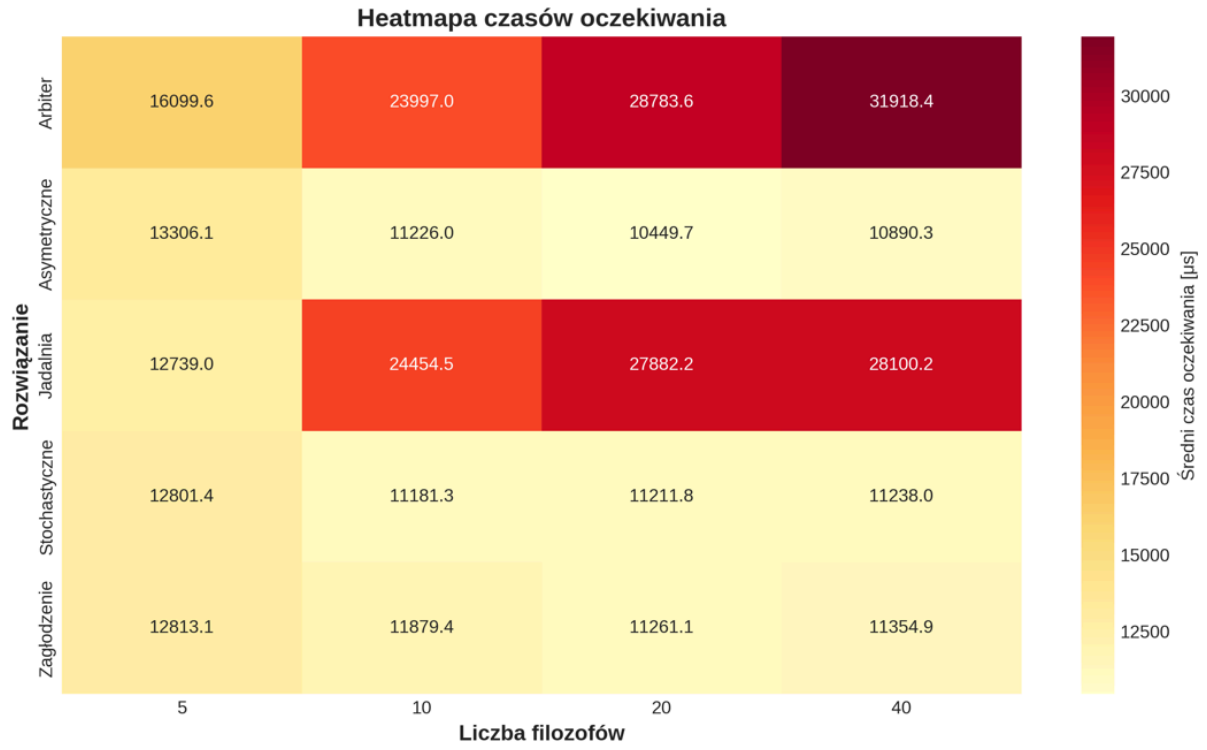
Rozwiązanie zostało oparte o **Semafora** i **synchronized**. Zapewnia brak deadlocka i dodatkowe bezpieczeństwo poprzez zmianę kolejności. Filozof próbuje dostać się do jadalni(tryAcquire()), jeśli mu się uda - synchronicznie podnosi najpierw lewy potem prawy, a jesli nie dostal sie do jadalni to synchroniczne podnosi widelce w odwrotnej kolejnosc, najpierw prawy potem lewy.

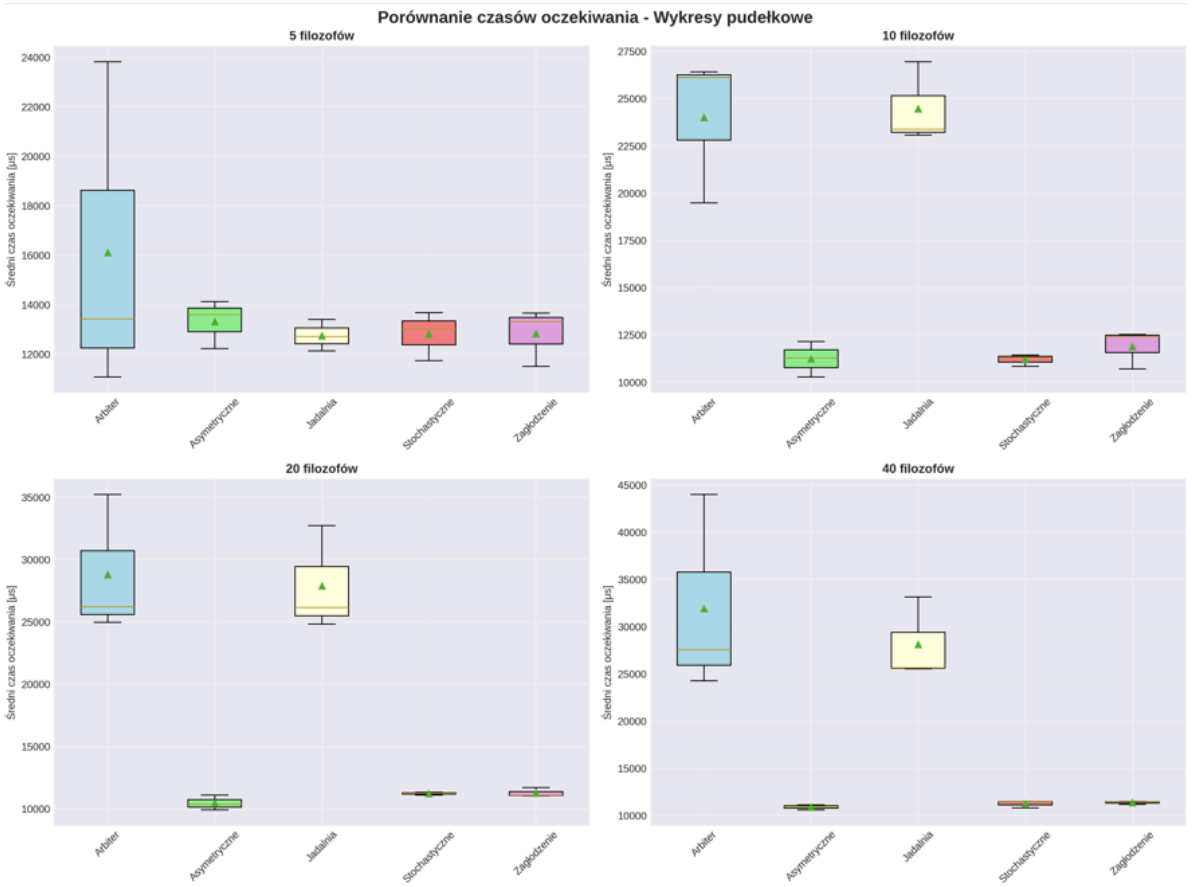
3. Analiza wyników działania programów

Analizę porównawczą poprzez rozszerzenie podstawowego kodu o dodatkowe zbieranie informacji, a następnie uzyskane informacje zwizualizowano przy użyciu matplotliba(python).

W celu uzyskania jak najlepszego porównania wykonano kilka uruchomień programu. W porównaniu pominięto metodę naiwną - jako iż nie jest w pełni poprawna. Poniższe wykresy przedstawiają czas oczekiwania filozofów na możliwość jedzenia. (Zagłodzenie z legendy oznacza rozwiązanie w wersji II - " z możliwym zagłodzeniem".





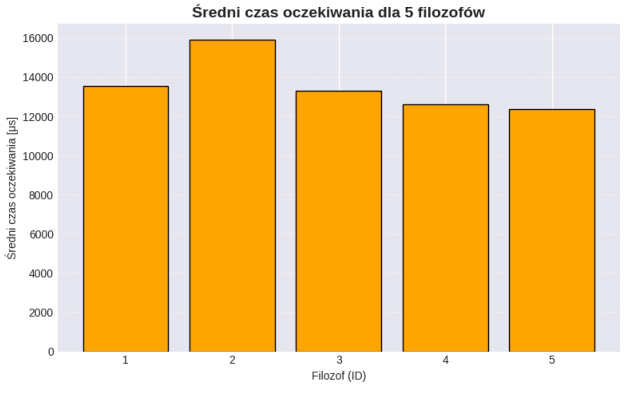


4. Porównanie wyników i wnioski

Na podstawie wyników możemy określić, że największy okres oczekiwania nastąpił dla algorytmów opartych na **semaforach**. Wyniki dla rozwiązań z jadalnią i arbitrem dla przypadków większej liczby filozofów są ponad dwukrotnie większe niż dla pozostałych metod. Dodatkowo rozwiązania z semaforami wykazują tendencje wzrostowe dla średnich czasów oczekiwania(im więcej filozofów tym więcej czasu oczekiwania), pozostałe algorytmy działają odwrotnie(im więcej filozofów tym mniejszy czas oczekiwania). Jest to zrozumiałe i spodziewane działanie, semafory są globalny synchronizatorem, muszą utrzymywać kolejkę wątków i ogólnie działają najwolniej. Dodatkowo występowanie dodatkowego koordynatora(arbitra) może powodować spowolnienia.

Dla małej ilości filozofów(<=5) rozwiązania wykazują podobne średnie czasy oczekiwania filozofów, we wszystkich próbach najszybciej wypadało rozwiązania z jadalnią, co wskazuje na efektywność tego podejścia dla małych danych.

Warto też zaznaczyć, że taka prezentacja danych na powyższych wykresach nie pokazuje wszystkich kluczowych momentów. Warto zauważyć, że tutaj na bardziej szczegółowym wykresie dla 5 filozofów dla rozwiązania z możliwością zagłodzenia występuje wspomniane zagłodzenie(dla filozofa 2), widac to po znacznie większym średnim



czasie oczekiwania. Jednakże umieszczanie tego typu wykresów dla każdej operacji uznałem za bezcelowe, pokazanie tego pojedynczego i ogólnych wykresów dla średnich wyczerpują możliwości odczytania najbardziej wartościowych skutków i zachowań z wykresów.

5. Instrukcja uruchomienia kodu

Ze względu na problemy sprzętowe realizacje projektu zaimplementowano w przeglądarkowym środowisku uruchomieniowym(ja używałem https://www.onlinegdb.com/online_java_compiler - wystarczy kleić tutaj zawartość pliku z kodem). Jednakże również bezpośrednie uruchomienie pliku konsolą powinno zadziałać(np. javac Main.java && java Main). Cały kod znajduje się w jednym pliku.

W metodzie main klasy Main znajduje się zmienna choice, którą możemy ustawić którą część rozwiązania uruchamiamy.