

Sprawozdanie lab 4

Autor: Seweryn Tasior

Wprowadzenie

Na laboratorium został zaprezentowany **Problem pięciu filozofów**. W ramach ćwiczenia zaimplementowano w Javie, 4 różne warianty tego problemu:

- Rozwiązanie naiwne (z możliwością blokady)
- Rozwiązanie z możliwością zagłodzenia
- Rozwiązanie asymetryczne
- Rozwiązanie z arbitrem Dodatkowo skorzystano z Pythona i bibliotek pandas, matplotlib, seaborn i numpy do narysowania wykresów porównawczych

W ćwiczeniu korzystano ze środowiska IntelliJ IDEA Ultimate. W pisaniu kodu skorzystano z pomocy Copilota w trybie Agent, z użyciem modelu Claude Sonnet 3.5

Wykonanie

Zacząłem od zaimplementowania klas `Fork` i `NaivePhilosopher`. Każdy Filozof musi mieć w sobie referencję do dwóch obok siebie Widelców. Same widelce działają na zasadzie semaforów binarnych. Również każdy z filozofów wykonuje się niezależnie więc konieczne jest żeby działał w ramach jednego wątku.

Dla uzyskania losowości i uzyskania rzeczywistych wyników, dodałem metodę `think`, która na jakiś czas "usypia" filozofa. Dodatkowo każde zjedzenie posiłku też zajmują dodatkową ilość czasu. Metody `stopWaiting` i `startWaiting` służą do odmierzenia czasu wykonania, który wykorzystamy przy tworzeniu wykresów.

Metoda `run` symuluje naiwny algorytm działania, gdzie filozof próbuje zabrać najpierw lewy widelec, jeśli może, a potem prawy. Jeśli mu się udaje to je i odkłada je z powrotem. W przeciwnym wypadku czeka na ich zwolnienie

```
public abstract class NaivePhilosopher extends Thread {
    private int id;
    private Fork firstFork;
    private Fork secondFork;
    private int mealsEaten;
    private long totalWaitingTime;
    private long startWaitingTime;

    public NaivePhilosopher(int id, Fork firstFork, Fork secondFork) {
        this.id = id;
        this.firstFork = firstFork;
        this.secondFork = secondFork;
        this.mealsEaten = 0;
        this.totalWaitingTime = 0;
    }

    public void think() throws InterruptedException {
        System.out.println("Filozof " + id + " myśli");
        Thread.sleep((long) (Math.random() * 100));
    }

    public void eat() throws InterruptedException {
        System.out.println("Filozof " + id + " je");
        mealsEaten++;
        Thread.sleep((long) (Math.random() * 100));
    }

    public void startWaiting() {
        startWaitingTime = System.currentTimeMillis();
    }

    public void stopWaiting() {
        totalWaitingTime += System.currentTimeMillis() - startWaitingTime;
    }

    public int getMealsEaten() {
        return mealsEaten;
    }

    public double getAverageWaitingTime() {
        return mealsEaten > 0 ? (double) totalWaitingTime / mealsEaten : 0;
    }

    @Override
    public void run() {
        try {
            while (!Thread.interrupted()) {
                think();

                startWaiting();
                System.out.println("Filozof " + id + " próbuje podnieść lewy widelec " + firstFork.getId());
                firstFork.take();
                System.out.println("Filozof " + id + " podniósł lewy widelec " + firstFork.getId());

                System.out.println("Filozof " + id + " próbuje podnieść prawy widelec " + secondFork.getId());
                secondFork.take();
                stopWaiting();
                System.out.println("Filozof " + id + " podniósł prawy widelec " + secondFork.getId());

                eat();
            }
        } catch (InterruptedException e) {
            // Ignoring InterruptedException in run method
        }
    }
}
```

```

        secondFork.putDown();
        System.out.println("Filozof " + id + " odlozył prawy widelec " + secondFork.getId());
        firstFork.putDown();
        System.out.println("Filozof " + id + " odlozył lewy widelec " + firstFork.getId());
    }
} catch (InterruptedException e) {
    Thread.currentThread().interrupt();
}
}

}

public class Fork {
    private final int id;
    private boolean taken = false;

    public Fork(int id) {
        this.id = id;
    }

    public synchronized void take() throws InterruptedException {
        while (taken) {
            wait();
        }
        taken = true;
    }

    public synchronized void putDown() {
        taken = false;
        notify();
    }

    public int getId() {
        return id;
    }

    public synchronized boolean isTaken() {
        return taken;
    }
}

```

Następnie zapytałem się modelu LLM o pomoc:

Chcę zaimplementować problem pięciu filozofów, zacznijmy od podejścia naiwnego. Pomóż wykonać eksperymenty dla różnej liczby filozofów. Dodatkowo zmierz średni czas oczekiwania każdego filozofa na dostęp do pałeczek. Wykonaj kilka pomiarów dla każdego przypadku testowego. Zaimplementuj rozwiązanie w `Main.java`

Jako wynik dostałem:

```

import java.util.Scanner;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;

public class Main {
    private static final int NUMBER_OF_PHILOSOPHERS = 5;
    private static final int SIMULATION_TIME = 10000; // 10 seconds
    private static final int NUMBER_OF_MEASUREMENTS = 5;

    public static void main(String[] args) {
        runNaiveSolution();
    }

    private static class MeasurementResult {
        final double[] waitingTimes;
        final int[] mealsEaten;

        MeasurementResult(double[] waitingTimes, int[] mealsEaten) {
            this.waitingTimes = waitingTimes;
            this.mealsEaten = mealsEaten;
        }
    }

    private static void runNaiveSolution() {
        Fork[] forks = new Fork[NUMBER_OF_PHILOSOPHERS];
        NaivePhilosopher[] philosophers = new NaivePhilosopher[NUMBER_OF_PHILOSOPHERS];

        // Initialize forks
        for (int i = 0; i < NUMBER_OF_PHILOSOPHERS; i++) {
            forks[i] = new Fork(i);
        }

        try {
            // Initialize philosophers
            for (int i = 0; i < NUMBER_OF_PHILOSOPHERS; i++) {
                Fork leftFork = forks[i];

```

```

        Fork rightFork = forks[(i + 1) % NUMBER_OF_PHILOSOPHERS];
        philosophers[i] = new NaivePhilosopher(i, leftFork, rightFork);
    }

    // Start philosophers
    for (NaivePhilosopher philosopher : philosophers) {
        philosopher.start();
    }

    // Let them run for SIMULATION_TIME
    Thread.sleep(SIMULATION_TIME);

    // Stop philosophers and collect results
    double[] waitingTimes = new double[NUMBER_OF_PHILOSOPHERS];
    int[] mealsEaten = new int[NUMBER_OF_PHILOSOPHERS];

    for (int i = 0; i < NUMBER_OF_PHILOSOPHERS; i++) {
        philosophers[i].interrupt();
        philosophers[i].join();
        waitingTimes[i] = philosophers[i].getAverageWaitingTime();
        mealsEaten[i] = philosophers[i].getMealsEaten();
        System.out.printf("Filozof %d - posilki: %d, sredni czas oczekiwania: %.2f ms%n",
            i, mealsEaten[i], waitingTimes[i]);
    }

    return new MeasurementResult(waitingTimes, mealsEaten);

} catch (Exception e) {
    e.printStackTrace();
    return new MeasurementResult(new double[NUMBER_OF_PHILOSOPHERS], new int[NUMBER_OF_PHILOSOPHERS]);
}
}
}

```

Następnie odałem odpowiednie foldery: `arbiter`, `asymetric`, `naive` i `thread_starvation` Kolejno napisałem prompta:

Korzystając z treści CW4.pdf zaimplementuj, w oddzielnych folderach kolejno: Rozwiązanie z możliwością zagłodzenia, Rozwiązanie asymetryczne Rozwiązanie z arbitrem. Skorzystaj z istniejącej klasy Fork. Budowę wzoruj na klasie `NaivePhilosopher`.

W wyniku otrzymałem 4 nowe klasy. Zauważyłem, że można stworzyć klasę abstrakcyjną `Philosopher`, która trzyma wspólne własności dla wszystkich filozofów tak, więc ją zaimplementowałem i lekko zmodyfikowałem kod wygenerowany przez AI:

```

public abstract class Philosopher extends Thread {
    protected int id;
    protected Fork firstFork;
    protected Fork secondFork;
    protected int mealsEaten;
    protected long totalWaitingTime;
    protected long startWaitingTime;

    protected Philosopher(int id, Fork firstFork, Fork secondFork) {
        this.id = id;
        this.firstFork = firstFork;
        this.secondFork = secondFork;
        this.mealsEaten = 0;
        this.totalWaitingTime = 0;
    }

    protected void think() throws InterruptedException {
        System.out.println("Filozof " + id + " myśli");
        Thread.sleep((long) (Math.random() * 1000));
    }

    protected void eat() throws InterruptedException {
        System.out.println("Filozof " + id + " je");
        mealsEaten++;
        Thread.sleep((long) (Math.random() * 1000));
    }

    protected void startWaiting() {
        startWaitingTime = System.currentTimeMillis();
    }

    protected void stopWaiting() {
        totalWaitingTime += System.currentTimeMillis() - startWaitingTime;
    }

    public int getMealsEaten() {
        return mealsEaten;
    }

    public double getAverageWaitingTime() {
        return mealsEaten > 0 ? (double) totalWaitingTime / mealsEaten : 0;
    }
}

```

```

public class Arbiter {
    private final int numberOfPhilosophers;
    private final boolean[] forks;
    private int eatingCount;

    public Arbiter(int numberOfPhilosophers) {
        this.numberOfPhilosophers = numberOfPhilosophers;
        this.forks = new boolean[numberOfPhilosophers];
        this.eatingCount = 0;
    }

    public synchronized boolean takeForks(int philosopherId) throws InterruptedException {
        // Sprawdzamy, czy filozof może jeść (maksymalnie n-1 filozofów może jeść jednocześnie)
        while (eatingCount >= numberOfPhilosophers - 1 ||
            forks[philosopherId] ||
            forks[(philosopherId + 1) % numberOfPhilosophers]) {
            wait();
        }

        // Przydzielamy widelce
        forks[philosopherId] = true;
        forks[(philosopherId + 1) % numberOfPhilosophers] = true;
        eatingCount++;
        return true;
    }

    public synchronized void putDownForks(int philosopherId) {
        forks[philosopherId] = false;
        forks[(philosopherId + 1) % numberOfPhilosophers] = false;
        eatingCount--;
        notifyAll();
    }
}

import com.sewery.Philosopher;

public class ArbiterPhilosopher extends Philosopher {
    private final Arbiter arbiter;

    public ArbiterPhilosopher(int id, Arbiter arbiter) {
        super(id, null, null);
        this.arbiter = arbiter;
    }

    @Override
    public void run() {
        try {
            while (!Thread.interrupted()) {
                think();

                System.out.println("Filozof " + id + " probuje podniesc widelce");
                startWaiting();
                arbiter.takeForks(id);
                stopWaiting();
                System.out.println("Filozof " + id + " podniosl widelce");

                eat();

                arbiter.putDownForks(id);
                System.out.println("Filozof " + id + " odlozyl widelce");
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}

package com.sewery.asymetric;

import com.sewery.Fork;
import com.sewery.Philosopher;

public class AsymmetricPhilosopher extends Philosopher {

    public AsymmetricPhilosopher(int id, Fork leftFork, Fork rightFork) {
        super(id, id % 2 == 0 ? leftFork : rightFork, id % 2 == 0 ? rightFork : leftFork);
    }

    @Override
    public void run() {
        try {
            while (!Thread.interrupted()) {
                think();

                startWaiting();
                System.out.println("Filozof " + id + " probuje podniesc pierwszy widelec " + firstFork.getId());
                firstFork.take();
                System.out.println("Filozof " + id + " podniosl pierwszy widelec " + firstFork.getId());

                System.out.println("Filozof " + id + " probuje podniesc drugi widelec " + secondFork.getId());
            }
        }
    }
}

```

```

        secondFork.take();
        stopWaiting();
        System.out.println("Filozof " + id + " podniosl drugi widelec " + secondFork.getId());

        eat();

        secondFork.putDown();
        System.out.println("Filozof " + id + " odlozyl drugi widelec " + secondFork.getId());
        firstFork.putDown();
        System.out.println("Filozof " + id + " odlozyl pierwszy widelec " + firstFork.getId());
    }
} catch (InterruptedException e) {
    Thread.currentThread().interrupt();
}
}

package com.sewery.naive;

import com.sewery.Fork;
import com.sewery.Philosopher;

public class NaivePhilosopher extends Philosopher {

    public NaivePhilosopher(int id, Fork leftFork, Fork rightFork) {
        super(id, leftFork, rightFork);
    }

    @Override
    public void run() {
        try {
            while (!Thread.interrupted()) {
                think();

                startWaiting();
                System.out.println("Filozof " + id + " probuje podniesc lewy widelec " + firstFork.getId());
                firstFork.take();
                System.out.println("Filozof " + id + " podniosl lewy widelec " + firstFork.getId());

                System.out.println("Filozof " + id + " probuje podniesc prawy widelec " + secondFork.getId());
                secondFork.take();
                stopWaiting();
                System.out.println("Filozof " + id + " podniosl prawy widelec " + secondFork.getId());

                eat();

                secondFork.putDown();
                System.out.println("Filozof " + id + " odlozyl prawy widelec " + secondFork.getId());
                firstFork.putDown();
                System.out.println("Filozof " + id + " odlozyl lewy widelec " + firstFork.getId());
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}

package com.sewery.thread_starvation;

import com.sewery.Fork;
import com.sewery.Philosopher;

public class StarvationPhilosopher extends Philosopher {
    private static final Object globalLock = new Object();

    public StarvationPhilosopher(int id, Fork leftFork, Fork rightFork) {
        super(id, leftFork, rightFork);
    }

    @Override
    public void run(){
        Fork leftFork = firstFork;
        Fork rightFork = secondFork;
        try {
            while (!Thread.interrupted()) {
                think();

                startWaiting();
                synchronized (globalLock) {
                    System.out.println("Filozof " + id + " probuje podniesc widelce");

                    while (leftFork.isTaken() || rightFork.isTaken()) {
                        globalLock.wait();
                    }

                    leftFork.take();
                    rightFork.take();
                    stopWaiting();
                    System.out.println("Filozof " + id + " podniosl oba widelce");
                }
            }
        }
    }
}

```

```

        eat();

        synchronized (globalLock) {
            leftFork.putDown();
            rightFork.putDown();
            System.out.println("Filozof " + id + " odlozyl widelce");
            globalLock.notifyAll();
        }
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }
}
}

```

Rozwiązanie asymtetryczne, w porównaniu z naiwnym, różni się tylko innym doborem wideleca, który filozof będzie brał jako pierwszy

Wygenerowane przez czat rozwiązania z arbitrem i możliwością zgłoszenia sprawy, wywołały we mnie wątpliwości.

Wytłumaczenie rozwiązania z możliwością zagłodzenia

Algorytm zagłodzenia zakłada, że filozof oczekuje kiedy zwolnią się dwa widelece obok niego, by je jednocześnie oba zabrać. Klasa `StarvationPhilosopher` korzysta z statycznego globalnego locka, który ma za zadanie blokować dostęp każdego obiektu tej klasy do sekcji krytycznej. To znaczy, że sekcję `synchronized (globalLock)` może wykonywać tylko jeden filzof, dzięki czemu jesteśmy pewni że jeden filozof w danym momencie podnosi albo odkłada dwa widelce (albo pałeczki) naraz. Jednak jest pewien wyjątek, gdy wątek niespełni warunek poniższej pętli:

```

while (leftFork.isTaken() || rightFork.isTaken()) {
    globalLock.wait();
}

```

To przechodzi on w stan `interrupted`, i blokada `globalLock` jest znoszona. Ci filozofowie są budzeni poprze wywołanie `globalLock.notifyAll()`, gdy dwa widelece zostaną odłożone na miejsce

Wytłumaczenie rozwiązania z arbitrem

Rozwiązanie z Arbitrem nie korzysta z klasy `Fork`. Najważniejszą metodami są `takeForks` i `putDownForks`. Kiedy pierwsza z nich zostanie wywołana, `Arbiter` czeka z przydzieleniem mu widelców dopóki mniej niż `N-1` filozfów nie je oraz gdy widelece obok są wolne. Klasa `Arbiter` funkcjonalnie przypomina semafor z licznikiem

Tworzenie eksperymentów

Następnie Agent dostał ode mnie kolejne polecenie:

```

Analogiczne do rozwiązania naiwnego, napisz eksperymenty wykonujące zaimplementowane wcześniej algorytmy

```

Tym razem dostałem dużo niepotrzebnego kodu od LLMA. Po refaktoryzacji m.in. dodaniu prostej refleksji i generycznych paramterów by ujednolicić, testowanie rozwiązań oraz zapisywanie do pliku csv przez metodę `saveResults` otrzymałem:

```

package com.sewery;

import com.sewery.naive.NaivePhilosopher;
import com.sewery.thread_starvation.StarvationPhilosopher;
import com.sewery.asymetric.AsymmetricPhilosopher;
import com.sewery.arbiter.Arbiter;
import com.sewery.arbiter.ArbiterPhilosopher;
import java.util.Scanner;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;

public class Main {
    private static final int NUMBER_OF_PHILOSOPHERS = 5;
    private static final int SIMULATION_TIME = 10000; // 10 seconds
    private static final int NUMBER_OF_MEASUREMENTS = 5;

    public static void main(String[] args) {
        System.out.println("Wybierz implementacje:");
        System.out.println("0. Rozwiazanie naiwne");
        System.out.println("1. Rozwiazanie z mozliwoscia zaglodzenia");
        System.out.println("2. Rozwiazanie asymetryczne");
        System.out.println("3. Rozwiazanie z arbitrem");
        System.out.println("4. Uruchom wszystkie testy");

        Scanner scanner = new Scanner(System.in);
        int choice = scanner.nextInt();
        String csvFileName = "wyniki.csv";

        try (PrintWriter writer = new PrintWriter(new FileWriter(csvFileName))) {
            writer.println("Implementation,Measurement,Philosopher,WaitingTime,MealsEaten");

```

```

        if (choice == 4) {
            System.out.println("\n=== Wykonywanie pomiarow dla wszystkich implementacji ===\n");
            for (int i = 0; i < 4; i++) {
                runMeasurements(i, writer);
            }
        } else {
            runMeasurements(choice, writer);
        }

        System.out.println("\nWyniki zostaly zapisane do pliku: " + csvFileName);
    } catch (IOException e) {
        System.err.println("Blad podczas zapisu do pliku CSV: " + e.getMessage());
    }
}

private static void runMeasurements(int implementationChoice, PrintWriter writer) {
    String implementationName = switch (implementationChoice) {
        case 0 -> "Naiwne";
        case 1 -> "Z zaglodzeniem";
        case 2 -> "Asymetryczne";
        case 3 -> "Z arbitrem";
        default -> "Nieznana implementacja";
    };

    System.out.println("\n=== " + implementationName + " ===");
    System.out.println("Wykonywanie " + NUMBER_OF_MEASUREMENTS + " pomiarow...\n");

    for (int i = 0; i < NUMBER_OF_MEASUREMENTS; i++) {
        System.out.println("Pomiar " + (i + 1) + ":");
        switch (implementationChoice) {
            case 0 -> saveResults(runPhilosopherSolution(NaivePhilosopher.class), implementationName, i + 1, writer);
            case 1 -> saveResults(runPhilosopherSolution(StarvationPhilosopher.class), implementationName, i + 1, writer);
            case 2 -> saveResults(runPhilosopherSolution(AsymmetricPhilosopher.class), implementationName, i + 1, writer);
            case 3 -> saveResults(runArbiterSolution(), implementationName, i + 1, writer);
        }
        System.out.println();
    }
}

private static void saveResults(MeasurementResult result, String implementation, int measurementNumber, PrintWriter writer) {
    for (int i = 0; i < NUMBER_OF_PHILOSOPHERS; i++) {
        writer.printf("%s,%d,%d,%.2f,%d\n",
            implementation,
            measurementNumber,
            i,
            result.waitingTimes[i],
            result.mealsEaten[i]
        );
    }
    writer.flush();
}

private static class MeasurementResult {
    final double[] waitingTimes;
    final int[] mealsEaten;

    MeasurementResult(double[] waitingTimes, int[] mealsEaten) {
        this.waitingTimes = waitingTimes;
        this.mealsEaten = mealsEaten;
    }
}

private static <T extends Philosopher> MeasurementResult runPhilosopherSolution(Class<T> philosopherClass) {
    Fork[] forks = new Fork[NUMBER_OF_PHILOSOPHERS];
    Philosopher[] philosophers = new Philosopher[NUMBER_OF_PHILOSOPHERS];

    // Initialize forks
    for (int i = 0; i < NUMBER_OF_PHILOSOPHERS; i++) {
        forks[i] = new Fork(i);
    }

    try {
        // Initialize philosophers
        for (int i = 0; i < NUMBER_OF_PHILOSOPHERS; i++) {
            Fork leftFork = forks[i];
            Fork rightFork = forks[(i + 1) % NUMBER_OF_PHILOSOPHERS];
            philosophers[i] = philosopherClass.getDeclaredConstructor(int.class, Fork.class, Fork.class)
                .newInstance(i, leftFork, rightFork);
        }

        // Start philosophers
        for (Philosopher philosopher : philosophers) {
            philosopher.start();
        }

        // Let them run for SIMULATION_TIME
        Thread.sleep(SIMULATION_TIME);

        // Stop philosophers and collect results
    }
}

```

```

        double[] waitingTimes = new double[NUMBER_OF_PHILOSOPHERS];
        int[] mealsEaten = new int[NUMBER_OF_PHILOSOPHERS];

        for (int i = 0; i < NUMBER_OF_PHILOSOPHERS; i++) {
            philosophers[i].interrupt();
            philosophers[i].join();
            waitingTimes[i] = philosophers[i].getAverageWaitingTime();
            mealsEaten[i] = philosophers[i].getMealsEaten();
            System.out.printf("Filozof %d - posilki: %d, sredni czas oczekiwania: %.2f ms%n",
                               i, mealsEaten[i], waitingTimes[i]);
        }

        return new MeasurementResult(waitingTimes, mealsEaten);

    } catch (Exception e) {
        e.printStackTrace();
        return new MeasurementResult(new double[NUMBER_OF_PHILOSOPHERS], new int[NUMBER_OF_PHILOSOPHERS]);
    }
}

private static MeasurementResult runArbiterSolution() {
    Philosopher[] philosophers = new Philosopher[NUMBER_OF_PHILOSOPHERS];
    Arbiter arbiter = new Arbiter(NUMBER_OF_PHILOSOPHERS);

    // Initialize forks
    try {
        // Initialize philosophers
        for (int i = 0; i < NUMBER_OF_PHILOSOPHERS; i++) {
            philosophers[i] = new ArbiterPhilosopher(i, arbiter);
        }

        // Start philosophers
        for (Philosopher philosopher : philosophers) {
            philosopher.start();
        }

        // Let them run for SIMULATION_TIME
        Thread.sleep(SIMULATION_TIME);

        // Stop philosophers and collect results
        double[] waitingTimes = new double[NUMBER_OF_PHILOSOPHERS];
        int[] mealsEaten = new int[NUMBER_OF_PHILOSOPHERS];

        for (int i = 0; i < NUMBER_OF_PHILOSOPHERS; i++) {
            philosophers[i].interrupt();
            philosophers[i].join();
            waitingTimes[i] = philosophers[i].getAverageWaitingTime();
            mealsEaten[i] = philosophers[i].getMealsEaten();
            System.out.printf("Filozof %d - posilki: %d, sredni czas oczekiwania: %.2f ms%n",
                               i, mealsEaten[i], waitingTimes[i]);
        }

        return new MeasurementResult(waitingTimes, mealsEaten);

    } catch (Exception e) {
        e.printStackTrace();
        return new MeasurementResult(new double[NUMBER_OF_PHILOSOPHERS], new int[NUMBER_OF_PHILOSOPHERS]);
    }
}
}

```

Rysowanie wykresów

Na podstawie pliku z CSV postanowiłem napisać program z pomocą LLM do wizualizacji wykresów. Użyłem prompta:

Zaimplementuj w Pythonie program, wizualizacji danych z pliku `wyniki.csv`. Chciałbym zobaczyć wykresy pudełkowe. Zależy mi na porównaniu Czasów oczekiwania pomiędzy różnymi implementacjami, oraz ilość posiłków filozofa jakie każda z nich uzyskuje. Bądź zwięzły, nie dodawaj error checków, timestampów oraz niepotrzebnych stylów

Po kilku własnych mniejszych poprawkach uzyskałem kod:

```

import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np

def create_visualizations(df: pd.DataFrame):
    sns.set_theme(style="whitegrid")
    fig, axes = plt.subplots(3, 1, figsize=(15, 15), constrained_layout=True)

    # 1. Box Plot (Wykres pudełkowy)
    ax1 = axes[0]
    sns.boxplot(x='Implementacja', y='CzasOczekiwania', data=df, ax=ax1)
    # Uprozczone stylowanie
    ax1.set_title('Czasy oczekiwania dla różnych implementacji')

```



```
ax1.set_xlabel('Implementacja')
ax1.set_ylabel('Czas oczekiwania (ms)')
ax1.set_xticklabels(ax1.get_xticklabels(), rotation=45)

# Dodawanie wartości median
medians = df.groupby('Implementacja')['CzasOczekiwania'].median()
plot_categories = [t.get_text() for t in ax1.get_xticklabels()]
for i, category in enumerate(plot_categories):
    median_val = medians.loc[category]
    ax1.text(
        i, median_val, f'{median_val:.1f}',
        horizontalalignment='center', verticalalignment='bottom'
    )

# 2. Violin Plot (Wykres skrzypcowy)
ax2 = axes[1]
sns.violinplot(x='Implementacja', y='CzasOczekiwania', data=df, ax=ax2)
# Uproszczone stylowanie
ax2.set_title('Rozkład czasów oczekiwania')
ax2.set_xlabel('Implementacja')
ax2.set_ylabel('Czas oczekiwania (ms)')
ax2.set_xticklabels(ax2.get_xticklabels(), rotation=45)

# 3. Bar Plot (Wykres słupkowy)
ax3 = axes[2]
meals_stats = df.groupby('Implementacja')['PosilkiZjedzone'].agg(['mean', 'std']).reset_index()

bars = ax3.bar(meals_stats['Implementacja'], meals_stats['mean'])
ax3.errorbar(
    meals_stats['Implementacja'], meals_stats['mean'],
    yerr=meals_stats['std'], fmt='none', color='black', capsize=5
)
# Uproszczone stylowanie
ax3.set_title('Średnia liczba posiłków dla różnych implementacji')
ax3.set_xlabel('Implementacja')
ax3.set_ylabel('Średnia liczba posiłków')
ax3.set_xticklabels(ax3.get_xticklabels(), rotation=45)

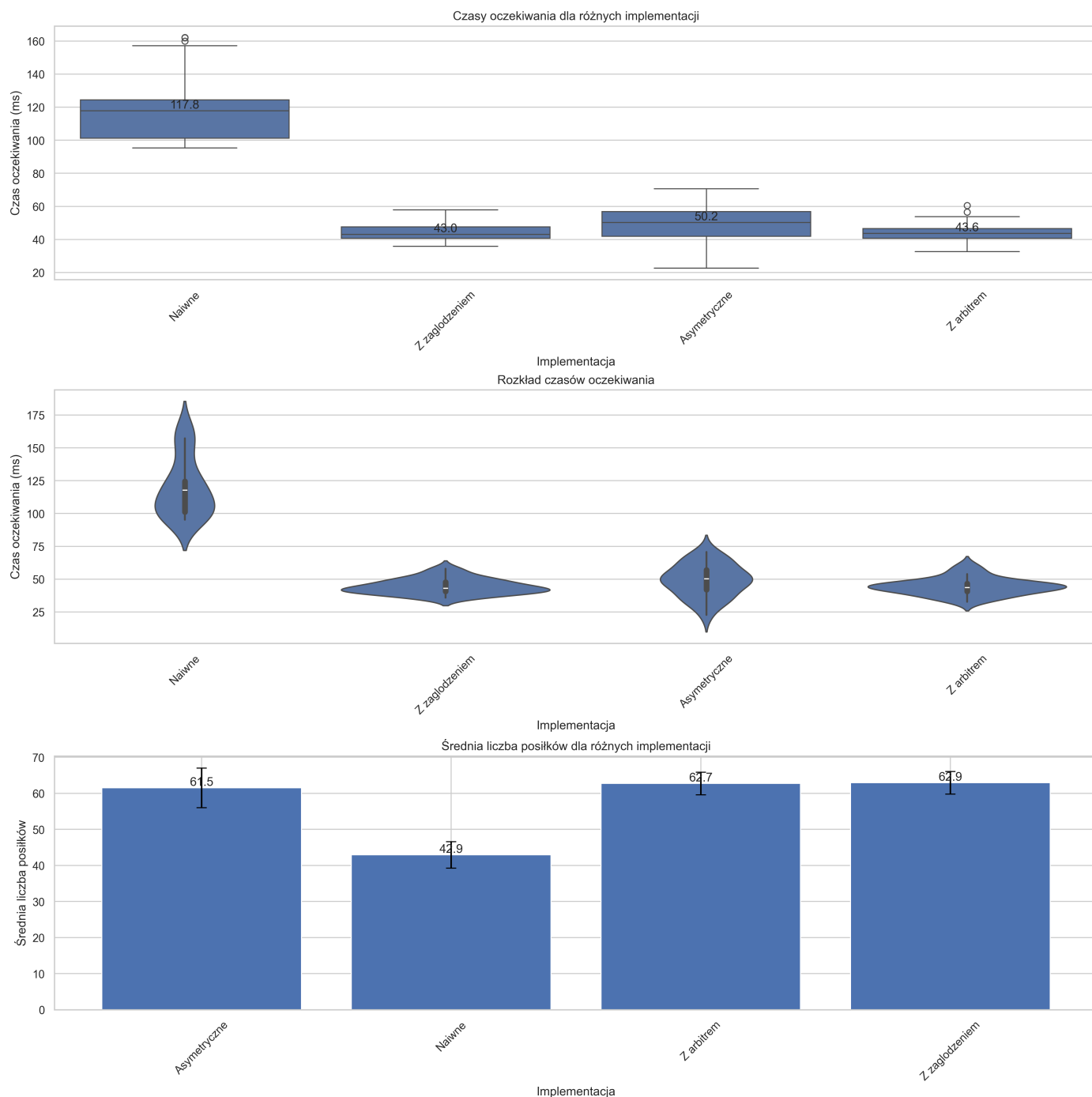
for bar in bars:
    height = bar.get_height()
    ax3.text(
        bar.get_x() + bar.get_width() / 2., height,
        f'{height:.1f}', ha='center', va='bottom'
    )

plt.savefig('philosophers_analysis.png', dpi=300, bbox_inches='tight')

def main():
    # Usunięto blok try...except
    df = pd.read_csv("wyniki.csv")

    create_visualizations(df)
if __name__ == "__main__":
    main()
```

Wizualizacja



Wnioskując z wykresu czasu oczekiwania filozofa na zwolnienie się widelców w implementacji Naïwej jest najbardziej rozbieżny oraz jego mediana jest największa (117.7 ms). Rozwiązanie z zagłóceniem radzi sobie wyjątkowo dobrze - wynika to z bardzo niskiego prawdopodobieństwa wystąpienia sytuacji w której jeden wątek ciągle oczekuje, na swoją kolej. W wynikach możemy dostrzec, że rozwiązanie z arbitrem ma mniejszą rozbieżność czasową, oraz mediana czas oczekiwania jest mniejsza aniżeli asymetrycznego. Tak więc najgorzej w zestawieniu wypada algorytm naiwny, natomiast najlepiej rozwiązanie asymetryczne i z zagłóceniem.

Podsumowanie

Eksperymenty z problemem pięciu filozfów, pokazały wpływ doboru algorytmu na działanie programu. Złe zaimplementowany program może się zakleszczyć, zagłodzić albo być niewydajny. Wybranie alogrytmu asymetrycznego daje w tym przypadku ochronę przed wyżej wymienionymi zagrożeniami.