Laboratorium Nr 6
"Problem pięciu filozofów"
Radosław Kopeć
10.11.2020

I. Zadanie 1

1. Treść zadania

Problem pięciu filozofów:

1. Każdy filozof zajmuje się głównie myśleniem
2. Od czasu do czasu potrzebuje zjeść
3. Do jedzenie potrzebne mu są oba widelce po jego prawej i lewej stronie
4. Jedzenie trwa skończona (ale nieokreślona z góry) ilość czasu, po czym filozof widelce odkłada i wraca do myślenia
5. Cykl powtarza się od początku

a. Zaimplementować trywialne rozwiązanie z symetrycznymi filozofami. Zaobserwować problem blokady.

b. Zaimplementować rozwiązanie z widelcami podnoszonymi jednocześnie. Jaki problem może tutaj wystąpić ?

c. Zaimplementować rozwiązanie z lokajem.

d. Wykonać pomiary dla każdego rozwiązania i wywnioskować co ma wpływ na wydajność każdego rozwiązania

2. Koncepcja rozwiązania

Symetyczni filozofowie:
Każdy z widelców posiada swój semafor, filozofowie podnoszą widelce w kolejności lewy, prawy, następnie jedzą i odkładają widelce w kolejności prawy,lewy.

Widelce podnoszone jednocześnie:
Operacja podnoszenia widelca działa teraz nieblokująco. Gdy filozofowi nie uda się podnieść dwóch widelców to odkłada jeden widelec (jeśli go trzyma) i próbuje ponownie za jakiś czas.

Rozwiązanie z lokajem:

Filozof przed rozpoczęciem jedzenia prosi kelnera o talerz. Kelner posiada semafor o wartości 4 reprezentujący ilość dostępnych talerzy. Maksymalnie 4 filozofów na raz może mieć talerz, dlatego nie dojdzie do zakleszczenia. Po zakończeniu jedzenia filozof odkłada talerz.

3. **Implementacja i wyniki:**

Implementacje zawierające wyłącznie kluczowe elementy implementacji:

Symetryczni filozofowie:

```
public class Fork {

    Semaphore semaphore = new Semaphore( permits: 1);

    public Fork(){}

    public void put() { semaphore.release(); }

    public void take(){
        try {
            semaphore.acquire();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

```
public Philosopher(Fork leftFork, Fork rightFork, String name, int foodAtPlate,int thinkTimeMs,int eatTimeMs,String color) {
    super();
    this.leftFork = leftFork;
    this.rightFork = rightFork;
    this.name = name;
    this.foodAtPlate = foodAtPlate;
    this.dinnerSize = foodAtPlate;
    this.thinkTimeMs = thinkTimeMs;
    this.eatTimeMs = eatTimeMs;
    this.COLOR = color;
}
```

```java
private void think(){
    try {
        sleep(thinkTimeMs);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

```java
private void eat(){

    try {
        Thread.sleep(eatTimeMs);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    System.out.println(COLOR + name + " ate " + (dinnerSize - foodAtPlate) + " time " + ANSI_RESET);

    foodAtPlate --;

}
```

```java
@Override
public void run(){

    while (!isDinnerFinished()){

        think();

        leftFork.take();
        rightFork.take();

        eat();

        leftFork.put();
        rightFork.put();

    }

    sayGoodbye();

}
```

Widelce podnoszone jednocześnie:

Fork:

```
public class Fork {

    Semaphore semaphore = new Semaphore( permits: 1);

    public Fork(){}

    public void put() { semaphore.release(); }

    public boolean take(){
        return semaphore.tryAcquire();
    }
}
```

Philosopher:

```
@Override
public void run(){

    while (!isDinnerFinished()){

        think();

        boolean hungry = true;
        while(hungry) {
            if (leftFork.take()) {
                if (rightFork.take()) {
                    eat();
                    hungry = false;
                    rightFork.put();
                }
                leftFork.put();
            }
        }
    }

    sayGoodbye();

}
```

Rozwiązanie z lokajem:

Waiter:

```
public class Waiter {

    Semaphore platesSemaphore;

    public Waiter(int philosophersAmount) { this.platesSemaphore = new Semaphore( permits: philosophersAmount - 1); }

    public void askAboutPlate(){
        try {
            platesSemaphore.acquire();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

    }

    public void realisePlate() { platesSemaphore.release(); }
}
```

Philosopher:

```
@Override
public void run(){

    while (!isDinnerFinished()){

        think();

        waiter.askAboutPlate();
        leftFork.take();
        rightFork.take();

        eat();

        rightFork.put();
        leftFork.put();
        waiter.realisePlate();

    }

    sayGoodbye();

}
```

4. Sposób pomiaru

Utworzenie pomiaru dla różnych rozwiązań w formie klasy implementującej interfejs Callable. Pomiary będą przeprowadzane dla zmiany trzech różnych parametrów:

- czasu myślenia
- czasu jedzenia
- ilości posiłków do zjedzenia

Wszystkie pojedyncze pomiary zostaną wykonane przez pule stu wątków w ExecutorService co pozwoli szybko pozyskać wyniki. Zakleszczenia zostaną rozpoznane poprzez wykonanie funkcji get() na obiekcie typu Future z ustawionym czasem timeoutu.

Symetryczni filozofowie:

```
public class SymmetricPhilosophers implements Callable<Long> {
    int dinnerSize;
    int eatTimeMs;
    int thinkTImeMs;

    public SymmetricPhilosophers(int dinnerSize, int eatTimeMs, int thinkTImeMs) {
        this.dinnerSize = dinnerSize;
        this.eatTimeMs = eatTimeMs;
        this.thinkTImeMs = thinkTImeMs;
    }

    public Long  call() throws Exception{

        Fork fork1 = new Fork();
        Fork fork2 = new Fork();
        Fork fork3 = new Fork();
        Fork fork4 = new Fork();
        Fork fork5 = new Fork();

        Philosopher phil1 = new Philosopher(fork1,fork2, name: "First",dinnerSize,thinkTImeMs,eatTimeMs, color: "\u001B[36m");
        Philosopher phil2 = new Philosopher(fork2,fork3, name: "Second",dinnerSize,thinkTImeMs,eatTimeMs, color: "\u001B[34m");
        Philosopher phil3 = new Philosopher(fork3,fork4, name: "Third",dinnerSize,thinkTImeMs,eatTimeMs, color: "\u001B[37m");
        Philosopher phil4 = new Philosopher(fork4,fork5, name: "Fourth",dinnerSize,thinkTImeMs,eatTimeMs, color: "\u001B[32m");
        Philosopher phil5 = new Philosopher(fork5,fork1, name: "Fifth",dinnerSize,thinkTImeMs,eatTimeMs, color: "\u001B[31m");

        List<Thread> threads = new ArrayList<~>();
        threads.add(phil1);
        threads.add(phil2);
        threads.add(phil3);
        threads.add(phil4);
        threads.add(phil5);

        return measureTime(threads);
    }
```

Widelce podnoszone jednocześnie:

```
public class TwoForksPhilosophers implements Callable<Long> {
    int dinnerSize;
    int thinkTImeMs;
    int eatTimeMs;

    public TwoForksPhilosophers(int dinnerSize, int thinkTImeMs, int eatTimeMs) {
        this.dinnerSize = dinnerSize;
        this.thinkTImeMs = thinkTImeMs;
        this.eatTimeMs = eatTimeMs;
    }

    @Override
    public Long call() throws Exception {

        Fork fork1 = new Fork();
        Fork fork2 = new Fork();
        Fork fork3 = new Fork();
        Fork fork4 = new Fork();
        Fork fork5 = new Fork();

        Philosopher phil1 = new Philosopher(fork1,fork2, name: "First",dinnerSize,thinkTImeMs,eatTimeMs, color: "\u001B[36m");
        Philosopher phil2 = new Philosopher(fork2,fork3, name: "Second",dinnerSize,thinkTImeMs,eatTimeMs, color: "\u001B[34m");
        Philosopher phil3 = new Philosopher(fork3,fork4, name: "Third",dinnerSize,thinkTImeMs,eatTimeMs, color: "\u001B[37m");
        Philosopher phil4 = new Philosopher(fork4,fork5, name: "Fourth",dinnerSize,thinkTImeMs,eatTimeMs, color: "\u001B[32m");
        Philosopher phil5 = new Philosopher(fork5,fork1, name: "Fifth",dinnerSize,thinkTImeMs,eatTimeMs, color: "\u001B[31m");

        List<Thread> threads = new ArrayList<~>();
        threads.add(phil1);
        threads.add(phil2);
        threads.add(phil3);
        threads.add(phil4);
        threads.add(phil5);

        return measureTime(threads);
    }
```

Rozwiązanie z lokajem:

```
public class WaiterPhilosophers implements Callable<Long> {
     int dinnerSize;
     int eatTimeMs;
     int thinkTImeMs;

    public WaiterPhilosophers(int dinnerSize, int eatTimeMs, int thinkTImeMs) {
        this.dinnerSize = dinnerSize;
        this.eatTimeMs = eatTimeMs;
        this.thinkTImeMs = thinkTImeMs;
    }

    public Long call() throws Exception{

        Fork fork1 = new Fork();
        Fork fork2 = new Fork();
        Fork fork3 = new Fork();
        Fork fork4 = new Fork();
        Fork fork5 = new Fork();

        Waiter waiter = new Waiter( philosophersAmount: 5);

        Philosopher phil1 = new Philosopher(fork1,fork2, name: "First",dinnerSize,thinkTImeMs,eatTimeMs, color: "\u001B[36m",waiter);
        Philosopher phil2 = new Philosopher(fork2,fork3, name: "Second",dinnerSize,thinkTImeMs,eatTimeMs, color: "\u001B[34m",waiter);
        Philosopher phil3 = new Philosopher(fork3,fork4, name: "Third",dinnerSize,thinkTImeMs,eatTimeMs, color: "\u001B[37m",waiter);
        Philosopher phil4 = new Philosopher(fork4,fork5, name: "Fourth",dinnerSize,thinkTImeMs,eatTimeMs, color: "\u001B[32m",waiter);
        Philosopher phil5 = new Philosopher(fork5,fork1, name: "Fifth",dinnerSize,thinkTImeMs,eatTimeMs, color: "\u001B[31m",waiter);

        List<Thread> threads = new ArrayList<~>();
        threads.add(phil1);
        threads.add(phil2);
        threads.add(phil3);
        threads.add(phil4);
        threads.add(phil5);

        return measureTime(threads);
    }
```

Klasa która wykonuje główną funkcje programu, mierzy poszczególne przypadki, oraz rysuje wykresy:

```
public static void main(String[] args) {

    System.out.println("Measure Dinner change");
    List<List<Double>> dinnerChange = measureForParameters( dinnerSize: 1, thinkTimeMs: 10, eatTimeMs: 10, pollsSize: 100, measurementsSize: 100, changingValue: 0, step: 1);
    double[] symmetricDinner =   dinnerChange.get(0).stream().mapToDouble(Double::doubleValue).toArray();
    double[] twoForksDinner =   dinnerChange.get(1).stream().mapToDouble(Double::doubleValue).toArray();
    double[] waiterDinner =   dinnerChange.get(2).stream().mapToDouble(Double::doubleValue).toArray();
    double[] xDinner =   dinnerChange.get(3).stream().mapToDouble(Double::doubleValue).toArray();

    draw2DPlots(xDinner,symmetricDinner,xDinner,twoForksDinner,xDinner,waiterDinner, name: "Symmetric", name2: "Two Forks", name3: "Waiter", plotName: "Time in function of d

    System.out.println("Measure ThinkTime change");
    List<List<Double>> thinkTimeChange = measureForParameters( dinnerSize: 10, thinkTimeMs: 1, eatTimeMs: 10, pollsSize: 100, measurementsSize: 100, changingValue: 1, step: 1);
    double[] symmetricThinkTime =   thinkTimeChange.get(0).stream().mapToDouble(Double::doubleValue).toArray();
    double[] twoForksThinkTime =   thinkTimeChange.get(1).stream().mapToDouble(Double::doubleValue).toArray();
    double[] waiterThinkTime =   thinkTimeChange.get(2).stream().mapToDouble(Double::doubleValue).toArray();
    double[] xThinkTime =   thinkTimeChange.get(3).stream().mapToDouble(Double::doubleValue).toArray();

    draw2DPlots(xDinner,symmetricThinkTime,xThinkTime,twoForksThinkTime,xThinkTime,waiterThinkTime, name: "Symmetric", name2: "Two Forks", name3: "Waiter", plotName: "Time i

    System.out.println("Measure EatTime change");
    List<List<Double>> eatTimeChange = measureForParameters( dinnerSize: 10, thinkTimeMs: 10, eatTimeMs: 1, pollsSize: 100, measurementsSize: 100, changingValue: 2, step: 1);
    double[] symmetricEatTimeChange =   eatTimeChange.get(0).stream().mapToDouble(Double::doubleValue).toArray();
    double[] twoForksEatTimeChange =   eatTimeChange.get(1).stream().mapToDouble(Double::doubleValue).toArray();
    double[] waiterEatTimeChange =   eatTimeChange.get(2).stream().mapToDouble(Double::doubleValue).toArray();
    double[] xEatTimeChange =   eatTimeChange.get(3).stream().mapToDouble(Double::doubleValue).toArray();

    draw2DPlots(xEatTimeChange,symmetricEatTimeChange,xEatTimeChange,twoForksEatTimeChange,xEatTimeChange,waiterEatTimeChange, name: "Symmetric", name2: "Two Forks", name
}
```

Funkcja uruchamiająca pomiary równolegle przy użyciu puli wątków:

```
private static List<List<Double>> measureForParameters(int dinnerSize, int thinkTimeMs, int eatTimeMs, int pollsSize,int measurementsSize,int changingValue,int step){
    ExecutorService poolSymmetric = Executors.newFixedThreadPool(pollsSize);
    ExecutorService poolTwoForks = Executors.newFixedThreadPool(pollsSize);
    ExecutorService poolWaiter = Executors.newFixedThreadPool(pollsSize);

    List<Future<Long>> futuresTwoForks = new ArrayList<>();
    List<Future<Long>> futuresSymmetric = new ArrayList<>();
    List<Future<Long>> futuresWaiter = new ArrayList<>();

    List<Double> symmetricResult = new ArrayList<>();
    List<Double> twoForksResult = new ArrayList<>();
    List<Double> waiterResult = new ArrayList<>();
    List<Double> x = new ArrayList<>();

    for(int i=0;i<measurementsSize;i++){

        switch (changingValue) {
            case 0 -> dinnerSize += step;
            case 1 -> thinkTimeMs += step;
            case 2 -> eatTimeMs += step;
        }
```

```
        SymmetricPhilosophers symmetricPhilosophers = new SymmetricPhilosophers(dinnerSize,thinkTimeMs,eatTimeMs);
        TwoForksPhilosophers twoForksPhilosophers = new TwoForksPhilosophers(dinnerSize,thinkTimeMs,eatTimeMs);
        WaiterPhilosophers waiterPhilosophers = new WaiterPhilosophers(dinnerSize,thinkTimeMs,eatTimeMs);

        futuresSymmetric.add(poolSymmetric.submit(symmetricPhilosophers));
        futuresTwoForks.add(poolTwoForks.submit(twoForksPhilosophers));
        futuresWaiter.add(poolWaiter.submit(waiterPhilosophers));

        switch (changingValue) {
            case 0 -> x.add((double) dinnerSize);
            case 1 -> x.add((double) thinkTimeMs);
            case 2 -> x.add((double) eatTimeMs);
        }

    }

    try {
        Thread.sleep( millis: 10000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
```

```
    try {
        for(int i=0;i< measurementsSize;i++){
            try {symmetricResult.add((double) futuresSymmetric.get(i).get( timeout: 10,TimeUnit.SECONDS));
            }
            catch (TimeoutException e){
                symmetricResult.add(null);
                System.out.println("InfiniteLoop 1");
            }
            twoForksResult.add((double) futuresTwoForks.get(i).get());

            waiterResult.add((double) futuresWaiter.get(i).get());

            System.out.println("Progress: " + (double) (i + 1)/measurementsSize);
        }
    }
    catch (InterruptedException | ExecutionException e) {
        e.printStackTrace();
    }

    double max = Math.max(
            Math.max(symmetricResult.stream().filter(Objects::nonNull).max(Double::compareTo).orElse((double) 0),
                    twoForksResult.stream().filter(Objects::nonNull).max(Double::compareTo).orElse((double) 0)),
            waiterResult.stream().filter(Objects::nonNull).max(Double::compareTo).orElse((double) 0));

      to visualize infinite
    List finalResultOfSymmetric = Arrays.asList(symmetricResult.stream().map(a -> Objects.requireNonNullElseGet(a, () -> max * 1.5)).toArray());

    return List.of(finalResultOfSymmetric, twoForksResult, waiterResult, x);
}
```

Funkcja rysująca wykres:

```
private static void draw2DPlots(double[] arguments, double[] values, double[] arguments2, double[] values2, d

    Plot2DPanel plot = new Plot2DPanel();
    // add a line plot to the PlotPanel
    // blue

    double max = Math.max(
            Math.max(Arrays.stream(values).max().orElse( other: 0),
                    Arrays.stream(values2).max().orElse( other: 0)),
            Arrays.stream(values3).max().orElse( other: 0));

    double min = Math.min(
            Math.min(Arrays.stream(values).min().orElse( other: 0),
                    Arrays.stream(values2).min().orElse( other: 0)),
            Arrays.stream(values3).min().orElse( other: 0));

    //blue
    plot.addLinePlot(name, arguments, values);
    //red
    plot.addLinePlot(name2, arguments2, values2);
    // green
    plot.addLinePlot(name3, arguments3, values3);

    // set y-axis boundary
    plot.setFixedBounds( axe: 1,min,max);

    // put the PlotPanel in a JFrame, as a JPanel
    JFrame frame = new JFrame(plotName);
    frame.setContentPane(plot);
    frame.setVisible(true);
}
```

5. Wnioski

Wyniki pomiarów:

Niebieski wykres - Symetryczni filozofowie
Czerwony wykres - Widelce podnoszone jednocześnie
Zielony wykres - Rozwiązanie z lokajem

Wykres zależności czasu wykonania od ilości posiłków.

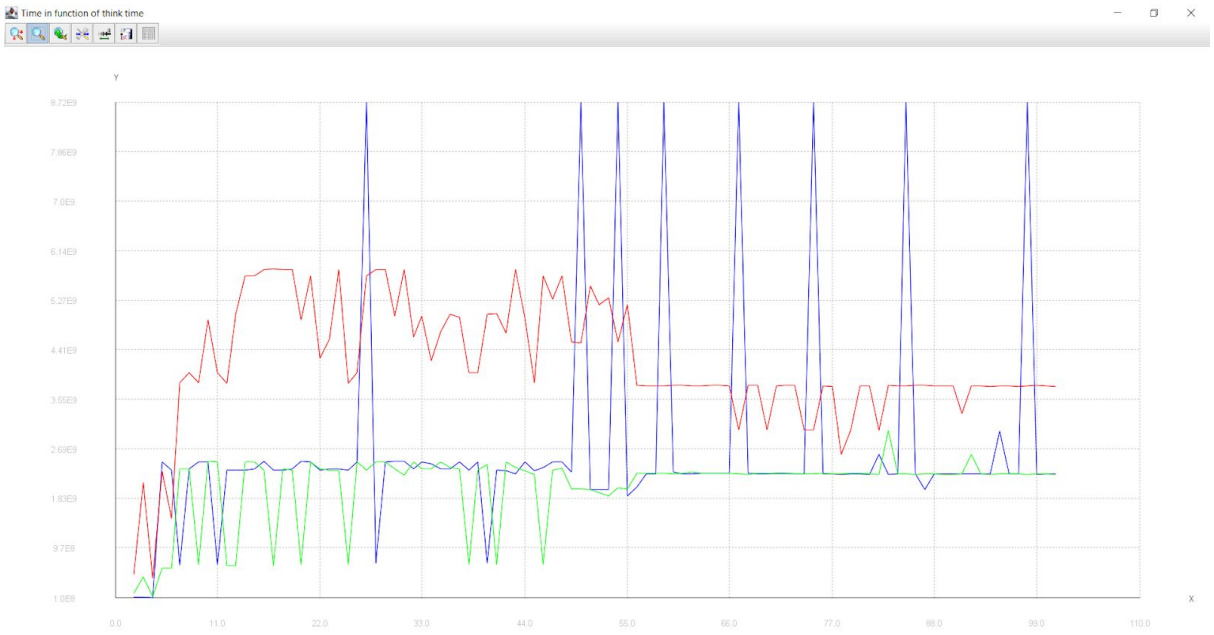
Niebieskie "piki" pokazują stan w którym dla symetrycznego rozwiązania nastąpiło zakleszczenie, czas dąży wtedy do nieskończoności. Można zauważyć wiele tego typu przypadków. Wniosek jaki się z tego wysuwa jest następujący:

- **Rozwiązanie z symetrycznymi filozofami powoduje zakleszczenia.**

Przeglądając charakterystyki wykresów dla rosnącej liczby posiłków, można zauważyć że czas również się zwiększa. Najlepiej wypada tutaj rozwiązanie z kelnerem ponieważ ma ono średnio najniższy czas, oraz nie powoduje zakleszczeń. Rozwiązanie z jednoczesnym podnoszeniem widelców nie powoduje zakleszczeń, ale przeprowadzenie tego samego problemu dla tych samych danych zajmuje mu znacząco więcej czasu niż przy użyciu rozwiązania z kelnerem. Rozwiązanie z symetrycznymi filozofami jest szybkie, porównywalnie, a nawet odrobinę szybciej niż z rozwiązanie z kelnerem jednak często prowadzi ono do zakleszczeń.

Niebieski wykres - Symetryczni filozofowie
Czerwony wykres - Widelce podnoszone jednocześnie
Zielony wykres - Rozwiązanie z lokajem

Wykres zależności czasu wykonania od czasu myślenia.



Niebieskie "piki" pokazują stan w którym dla symetrycznego rozwiązania nastąpiło zakleszczenie, czas dąży wtedy do nieskończoności. Można zauważyć wiele tego typu przypadków. Wniosek jaki się z tego wysuwa jest następujący:

- **Rozwiązanie z symetrycznymi filozofami powoduje zakleszczenia.**

Dla niskiego czasu myślenia, często dochodzi do rywalizacji co powoduje zwiększenie czasu dla czerwonego wykresu (przypadek z widelcami podnoszonymi jednocześnie). Gdy czas myślenia się zwiększa to filozofowie mają większe prawdopodobieństwo trafienia na wolny widelec, rzadziej dochodzi do wyścigu, widelce praktycznie stoją bezczynnie, objawia się to wyrównaniem charakterystyk wykresów. Najlepiej wypada tutaj rozwiązanie z kelnerem ponieważ ma ono średnio najniższy czas, oraz nie powoduje zakleszczeń. Rozwiązanie z jednoczesnym podnoszeniem widelców nie powoduje zakleszczeń, ale przeprowadzenie tego samego problemu dla tych samych danych zajmuje mu znacząco więcej czasu niż przy użyciu rozwiązania z kelnerem. Rozwiązanie z symetrycznymi filozofami jest szybkie jednak często prowadzi do zakleszczeń.

Niebieski wykres - Symetryczni filozofowie
Czerwony wykres - Widelce podnoszone jednocześnie
Zielony wykres - Rozwiązanie z lokajem

Wykres zależności czasu wykonania od czasu jedzenia.

Niebieskie "piki" pokazują stan w którym dla symetrycznego rozwiązania nastąpiło zakleszczenie, czas dąży wtedy do nieskończoności. Można zauważyć wiele tego typu przypadków. Wniosek jaki się z tego wysuwa jest następujący:

- **Rozwiązanie z symetrycznymi filozofami powoduje zakleszczenia.**

Podobnie jak w poprzednich przypadkach rozwiązania z kelnerem i z symetrycznymi filozofami są najszybsze, jednak pierwsze nie powoduje zakleszczeń. Rozwiązanie z podnoszonymi jednocześnie widelcami nie powoduje zakleszczeń, ale jest wolniejsze.

6. Bibliografia

http://home.agh.edu.pl/~funika/tw/lab6/
https://en.wikipedia.org/wiki/Dining_philosophers_problem