Teoria Współbieżności Raport 1 – Problem Ucztujących Filozofów.

Krzysztof Swędzioł

Część wspólna kodu dla wszystkich zadań (Widelec) :

```
class  Fork{
    public int number;
    public boolean taken;
    public Fork(int n){
        number = n;
        taken = false;
    }
    public synchronized void takeFork(){
        while(taken) {
            try {
                wait();
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }
        taken = true;
    }
    public synchronized void releaseFork(){
        taken = false;
        notifyAll();
    }
}
```

Każdemu widelcowi jak i filozofowi przyporządkowujemy odpowiedni numer, dzięki temu wiemy, który widelec dla danego filozofa jest lewy, a który prawy. Na przykład filozof numer 5 ma lewy widelec o numerze 5 i prawy widelec o numerze 6.

Zadanie 1

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

```
class Philosopher extends Thread{  2 usages                                    ⚠ 40 ⚠
    public int number;  7 usages
    public Fork leftFork;  5 usages
    public Fork rightFork;  5 usages
    public Philosopher(int n, Fork leftFork, Fork rightFork){  1 usage
        number = n;
        this.leftFork = leftFork;
        this.rightFork = rightFork;
    }
    public void think(){  1 usage
        try {
            System.out.println("Philosopher " + number + " is thinking...");
            //Thread.sleep(2000);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
    public void eat(){  1 usage
        try {
            System.out.println("Philosopher " + number + " is eating...");
            //Thread.sleep(2000);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
    public void takeLeft(){  1 usage
        System.out.println("Philosopher " + number + " is trying to take his left fork of num: "+ leftFork.number);
        this.leftFork.takeFork();
        System.out.println("Philosopher " + number + " has taken: "+ leftFork.number);

    }
```

```
    public void takeRight(){  1 usage
        System.out.println("Philosopher " + number + " is trying to take his right fork of num: "+ rightFork.number);
        this.rightFork.takeFork();
        System.out.println("Philosopher " + number + " has taken: "+ rightFork.number);

    }
    public void releaseLeft() { this.leftFork.releaseFork(); }
    public void releaseRight() { this.rightFork.releaseFork(); }


    public void run(){
        while(true){
            this.think();
            this.takeLeft();
            this.takeRight();
            this.eat();
            this.releaseLeft();
            this.releaseRight();

        }
    }

}
```

Zadanie 2

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 za głodzenie, gdyż oba widelce nigdy nie będą wolne.

W tym rozwiązaniu aby zasymulować jednoczesne wzięcie widelców postępuję następująco : filozof podnosi lewy widelec i sprawdza czy prawy jest wolny. Jeśli prawy jest zajęty to filozof nie czeka na niego tylko puszcza swój lewy widelec. Następnie ponownie próbuje podnieść lewy a następnie prawy i jeśli uda mu się podnieść prawy od razu, bez czekania, to przystępuje do jedzenia.

```java
class PhilosopherZad2 extends Thread{  2 usages
    public int number;  7 usages
    public Fork leftFork;  5 usages
    public Fork rightFork;  6 usages
    public boolean bothTaken = false;  2 usages
    public long totalWaitTime = 0;  4 usages
    public int eatAttempts = 0;  3 usages
    public PhilosopherZad2(int n, Fork leftFork, Fork rightFork){  1 usage
        number = n;
        this.leftFork = leftFork;
        this.rightFork = rightFork;
    }
    public void think(){...}
    public void eat(){  1 usage
        try {
            System.out.println("Philosopher " + number + " is eating...");
            Thread.sleep( millis: 500);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
    public void takeLeft(){  1 usage
        System.out.println("Philosopher " + number + " is trying to take his left fork of num: "+ leftFork.number);
        long startWait = System.nanoTime();
        this.leftFork.takeFork();
        long waitTime = System.nanoTime() - startWait;
        this.totalWaitTime += waitTime;
        System.out.println("Philosopher " + number + " has taken: "+ leftFork.number);
```

```java
public void takeRight(){  1 usage
    System.out.println("Philosopher " + number + " is trying to take his right fork of num: "+ rightFork.number);
    if(this.rightFork.taken == true){
        releaseLeft();
    }else{
        long startWait = System.nanoTime();
        this.rightFork.takeFork();
        long waitTime = System.nanoTime() - startWait;
        this.totalWaitTime += waitTime;
        this.bothTaken = true;
    }
    System.out.println("Philosopher " + number + " has taken: "+ rightFork.number);

}
public void releaseLeft() { this.leftFork.releaseFork(); }
public void releaseRight() { this.rightFork.releaseFork(); }

public double getAverageWaitTime() {  no usages
    if(eatAttempts != 0){
        return (totalWaitTime / (double) eatAttempts) / 1_000_000;
    }else{
        return this.totalWaitTime;
    }
}
```

```java
    public void run(){
        while(true){
            this.think();
            while(bothTaken != true){
                this.takeLeft();
                this.takeRight();
            }
            this.eat();
            this.eatAttempts++;
            this.releaseLeft();
            this.bothTaken = false;
            this.releaseRight();

        }
    }

}
```

Zadanie 3

Rozwiązanie asymetryczne. Filozofowie są ponumerowani. Filozof z parzystym numerem najpierw podnosi prawy widelec, filozof z nie parzystym numerem najpierw podnosi lewy widelec.

```
class PhilosopherZad3 extends Thread{  2 usages
    public int number;  8 usages
    public Fork leftFork;  5 usages
    public Fork rightFork;  5 usages
    public long totalWaitTime = 0;  4 usages
    public int eatAttempts = 0;  2 usages
    public PhilosopherZad3(int n, Fork leftFork, Fork rightFork){  1 usage
        number = n;
        this.leftFork = leftFork;
        this.rightFork = rightFork;
    }
    public void think(){  1 usage
        try {
            System.out.println("Philosopher " + number + " is thinking...");
            Thread.sleep( millis: 500);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
    public void eat(){  1 usage
        try {
            System.out.println("Philosopher " + number + " is eating...");
            Thread.sleep( millis: 500);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
```

```
public void takeLeft(){  2 usages
    System.out.println("Philosopher " + number + " is trying to take his left fork of num: "+ leftFork.number);
    long startWait = System.nanoTime();
    this.leftFork.takeFork();
    long waitTime = System.nanoTime() - startWait;
    this.totalWaitTime += waitTime;
    System.out.println("Philosopher " + number + " has taken: "+ leftFork.number);


}
public void takeRight(){  2 usages
    System.out.println("Philosopher " + number + " is trying to take his right fork of num: "+ rightFork.number);
    long startWait = System.nanoTime();
    this.rightFork.takeFork();
    long waitTime = System.nanoTime() - startWait;
    this.totalWaitTime += waitTime;
    System.out.println("Philosopher " + number + " has taken: "+ rightFork.number);

}
public void releaseLeft() { this.leftFork.releaseFork(); }
public void releaseRight() { this.rightFork.releaseFork(); }
```

```java
    public double getAverageWaitTime() {  no usages
        if(eatAttempts != 0){
            return (totalWaitTime / (double) eatAttempts) / 1_000_000;
        }else{
            return this.totalWaitTime;
        }
    }

    public void run(){
        while(true){
            this.think();
            if(this.number%2 == 0){
                this.takeRight();
                this.takeLeft();
            }else{
                this.takeLeft();
                this.takeRight();
            }
            this.eat();
            this.releaseLeft();
            this.releaseRight();

        }
    }

}
```

Zadanie 4

Rozwiązanie stochastyczne. Każdy filozof rzuca monetą tuż przed podniesieniem widelców i w ten sposób decyduje, który najpierw podnieść- lewy czy prawy (z prawdopodobieństwem 1 nie dojdzie do zagłodzenia).

```
class PhilosopherZad4 extends Thread{  2 usages
    public int number;  7 usages
    public Fork leftFork;  5 usages
    public Fork rightFork;  5 usages
    public long totalWaitTime = 0;  4 usages
    public int eatAttempts = 0;  3 usages
    public PhilosopherZad4(int n, Fork leftFork, Fork rightFork){  1 usage
        number = n;
        this.leftFork = leftFork;
        this.rightFork = rightFork;
    }
    public void think(){  1 usage
        try {
            System.out.println("Philosopher " + number + " is thinking...");
            Thread.sleep( millis: 500);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
    public void eat(){  1 usage
        try {
            System.out.println("Philosopher " + number + " is eating...");
            Thread.sleep( millis: 500);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
```

```java
public void takeLeft(){  2 usages
    System.out.println("Philosopher " + number + " is trying to take his left fork of num: "+ leftFork.number);
    long startWait = System.nanoTime();
    this.leftFork.takeFork();
    long waitTime = System.nanoTime() - startWait;
    this.totalWaitTime += waitTime;
    System.out.println("Philosopher " + number + " has taken: "+ leftFork.number);


}
public void takeRight(){  2 usages
    System.out.println("Philosopher " + number + " is trying to take his right fork of num: "+ rightFork.number);
    long startWait = System.nanoTime();
    this.rightFork.takeFork();
    long waitTime = System.nanoTime() - startWait;
    this.totalWaitTime += waitTime;
    System.out.println("Philosopher " + number + " has taken: "+ rightFork.number);

}
public void releaseLeft() { this.leftFork.releaseFork(); }
public void releaseRight() { this.rightFork.releaseFork(); }

public double getAverageWaitTime() {  no usages
    if(eatAttempts != 0){
        return (totalWaitTime / (double) eatAttempts) / 1_000_000;
    }else{
        return this.totalWaitTime;
    }
}
```

```java
    public void run(){
        while(true){
            this.think();
            double coinFlip = Math.random();
            if(coinFlip <= 0.5){
                this.takeLeft();
                this.takeRight();
            }else{
                this.takeRight();
                this.takeLeft();
            }
            this.eat();
            this.eatAttempts++;
            this.releaseLeft();
            this.releaseRight();

        }
    }

}
```

Zadanie 5

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.

W tym zadaniu jako arbitra wprowadziłem kelnera – jest to klasa Waiter, która w nieskończonej pętli sprawdza czy w danej chwili wystąpił deadlock (W tym przypadku jest on możliwy tylko wtedy, gdy każdy filozof trzyma swój lewy widelec), jeśli deadlock wystąpił, waiter losuje filozofa do powstrzymania, które wygląda w taki sposób że waiter wyciąga filozofowi z ręki jego widelec i kładzie na stole.

Waiter :

```
class Waiter extends Thread{  4 usages
    private List<PhilosopherZad5> philosophers = new ArrayList<>();  4 usages
    private List<Thread> philosopherThreads = new ArrayList<>();  1 usage
    private PhilosopherZad5 blockedFilosopher;  3 usages

    public synchronized void addPhilosopher(PhilosopherZad5 philosopher, Thread philosopherThread){  1 usage
        this.philosophers.add(philosopher);
        this.philosopherThreads.add(philosopherThread);
    }
    public synchronized void restrainPhilosopher(PhilosopherZad5 philosopher) { philosopher.waiterRestrain(); }
    public synchronized void releasePhilosopher(PhilosopherZad5 philosopher){  1 usage
        if(philosopher != null){
            if(philosopher.allowedToEat == false){
                philosopher.waiterRelease();
            }
        }
    }
    public synchronized boolean checkDeadlock(){  1 usage
        boolean flag = false;
        for(PhilosopherZad5 philosopher: philosophers){
            if(philosopher.getState() != Thread.State.WAITING){
                flag = false;
                return flag;
            }
        }
        flag = true;
        return flag;
    }
```

```
    public synchronized PhilosopherZad5 selectPhilosopher(){  1 usage
        int philosopherNum = (int) (Math.random() * philosophers.size());
        return this.philosophers.get(philosopherNum);
    }
    public synchronized void somePhilosopherFinishedEating(){  1 usage
        this.releasePhilosopher(this.blockedFilosopher);
    }
    public void run(){
        while(true){
            if(this.checkDeadlock()){
                this.blockedFilosopher = this.selectPhilosopher();
                this.restrainPhilosopher(this.blockedFilosopher);
            }
        }
    }
}
```

Filozof :

```
class PhilosopherZad5 extends Thread{  11 usages
    public int number;  9 usages
    public Fork leftFork;  5 usages
    public Fork rightFork;  6 usages
    public Waiter waiter;  2 usages
    public boolean allowedToEat = true;  4 usages
    public int hasBothForks = 0;  6 usages
    public long totalWaitTime = 0;  4 usages
    public int eatAttempts = 0;  3 usages

    public PhilosopherZad5(int n, Fork leftFork, Fork rightFork, Waiter waiter){  1 usage
        number = n;
        this.leftFork = leftFork;
        this.rightFork = rightFork;
        this.waiter = waiter;
    }
    public void think(){  1 usage
        try {
            System.out.println("Philosopher " + number + " is thinking...");
            Thread.sleep( millis: 500);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
```

```java
public void eat(){  1 usage
    try {
        System.out.println("Philosopher " + number + " is eating...");
        Thread.sleep( millis: 500);
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
public void takeLeft(){  1 usage
    System.out.println("Philosopher " + number + " is trying to take his left fork of num: "+ leftFork.number);
    long startWait = System.nanoTime();
    this.leftFork.takeFork();
    long waitTime = System.nanoTime() - startWait;
    this.totalWaitTime += waitTime;
    this.hasBothForks++;
    System.out.println("Philosopher " + number + " has taken his left fork: "+ leftFork.number);


}
public void takeRight(){  1 usage
    System.out.println("Philosopher " + number + " is trying to take his right fork of num: "+ rightFork.number);
    long startWait = System.nanoTime();
    this.rightFork.takeFork();
    long waitTime = System.nanoTime() - startWait;
    this.totalWaitTime += waitTime;
    this.hasBothForks++;
    System.out.println("Philosopher " + number + " has taken his right fork: "+ rightFork.number);

}
```

```java
    public void releaseLeft(){  2 usages
        this.leftFork.releaseFork();
        this.hasBothForks--;
    }
    public void releaseRight(){  1 usage
        this.rightFork.releaseFork();
        this.hasBothForks--;
    }

    public void waiterRestrain(){  1 usage
        System.out.println("Philosopher " + number + " is restrained");
        this.releaseLeft();
        this.allowedToEat = false;
    }
    public void waiterRelease(){  1 usage
        System.out.println("Philosopher " + number + " was released");
        this.allowedToEat = true;
    }
    public double getAverageWaitTime() {  no usages
        if(eatAttempts != 0){
            return (totalWaitTime / (double) eatAttempts) / 1_000_000;
        }else{
            return this.totalWaitTime;
        }
    }
```

```
    public void run(){
        while(true){
            this.think();
            if(this.allowedToEat){
                this.takeLeft();
                this.takeRight();
                if(this.hasBothForks == 2){
                    this.eat();
                    this.waiter.somePhilosopherFinishedEating();
                    this.eatAttempts++;
                    this.releaseLeft();
                    this.releaseRight();
                }else if(this.hasBothForks == 1){
                    this.rightFork.releaseFork();
                }
            }
        }
    }
}
```

Zadanie 6

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).

Waiter :

```
class WaiterZad6 extends Thread{  4 usages
    private List<PhilosopherZad6> philosophers = new ArrayList<>();  4 usages
    private List<Thread> philosopherThreads = new ArrayList<>();  1 usage
    private PhilosopherZad6 blockedFilosopher;  3 usages

    public synchronized void addPhilosopher(PhilosopherZad6 philosopher, Thread philosopherThread){  1 usage
        this.philosophers.add(philosopher);
        this.philosopherThreads.add(philosopherThread);
    }
    public synchronized void restrainPhilosopher(PhilosopherZad6 philosopher) { philosopher.waiterRestrain(); }
    public synchronized void releasePhilosopher(PhilosopherZad6 philosopher){  1 usage
        if(philosopher != null){
            if(philosopher.allowedToEat == false){
                philosopher.waiterRelease();
            }
        }
    }
    public synchronized boolean checkDeadlock(){  1 usage
        boolean flag = false;
        for(PhilosopherZad6 philosopher: philosophers){
            if(philosopher.getState() != Thread.State.WAITING){
                flag = false;
                return flag;
            }
        }
        flag = true;
        return flag;
    }
```

```
    public synchronized PhilosopherZad6 selectPhilosopher(){  1 usage
        int philosopherNum = (int) (Math.random() * philosophers.size());
        return this.philosophers.get(philosopherNum);
    }
    public synchronized void somePhilosopherFinishedEating(){  2 usages
        this.releasePhilosopher(this.blockedFilosopher);
    }
    public void run(){
        while(true){
            if(this.checkDeadlock()){
                this.blockedFilosopher = this.selectPhilosopher();
                this.restrainPhilosopher(this.blockedFilosopher);
            }
        }
    }
}
```

Filozof :

```java
class PhilosopherZad6 extends Thread {  11 usages
    public int number;  10 usages
    public Fork leftFork;  5 usages
    public Fork rightFork;  6 usages
    public WaiterZad6 waiter;  3 usages
    public boolean allowedToEat = true;  5 usages
    public int hasBothForks = 0;  7 usages
    public boolean hasToEatOutside = false;  5 usages
    public long totalWaitTime = 0;  4 usages
    private int eatAttempts = 0;  4 usages

    public PhilosopherZad6(int n, Fork leftFork, Fork rightFork, WaiterZad6 waiter) {  1 usage
        number = n;
        this.leftFork = leftFork;
        this.rightFork = rightFork;
        this.waiter = waiter;
    }

    public void think() {  1 usage
        try {
            System.out.println("Philosopher " + number + " is thinking...");
            Thread.sleep( millis: 500);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
```

```java
public void eat() {  2 usages
    try {
        if (this.hasToEatOutside) {
            System.out.println("Philosopher " + number + " is eating outside...");
        } else {
            System.out.println("Philosopher " + number + " is eating...");
        }
        Thread.sleep( millis: 500);
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}

public void takeLeft() {  2 usages
    System.out.println("Philosopher " + number + " is trying to take his left fork of num: " + leftFork.number);
    long startWait = System.nanoTime();
    this.leftFork.takeFork();
    long waitTime = System.nanoTime() - startWait;
    this.totalWaitTime += waitTime;
    this.hasBothForks++;
    System.out.println("Philosopher " + number + " has taken his left fork: " + leftFork.number);


}
```

```java
public void takeRight() {  2 usages
    System.out.println("Philosopher " + number + " is trying to take his right fork of num: " + rightFork.number);
    long startWait = System.nanoTime();
    this.rightFork.takeFork();
    long waitTime = System.nanoTime() - startWait;
    this.totalWaitTime += waitTime;
    this.hasBothForks++;
    System.out.println("Philosopher " + number + " has taken his right fork: " + rightFork.number);

}

public void releaseLeft() {  3 usages
    this.leftFork.releaseFork();
    this.hasBothForks--;
}

public void releaseRight() {  2 usages
    this.rightFork.releaseFork();
    this.hasBothForks--;
}

public void waiterRestrain() {  1 usage
    System.out.println("Philosopher " + number + " is restrained");
    this.releaseLeft();
    this.allowedToEat = false;
}
```

```java
public void waiterRelease() {  1 usage
    System.out.println("Philosopher " + number + " was released");
    this.allowedToEat = true;
}
public double getAverageWaitTime() {  no usages
    if(eatAttempts != 0){
        return (totalWaitTime / (double) eatAttempts) / 1_000_000;
    }else{
        return this.totalWaitTime;
    }
}
```

```
    public void run() {
        while (true) {
            this.think();
            if (this.allowedToEat && this.hasToEatOutside == false) {
                this.takeLeft();
                this.takeRight();
                if (this.hasBothForks == 2) {
                    this.eat();
                    this.waiter.somePhilosopherFinishedEating();
                    this.eatAttempts++;
                    this.releaseLeft();
                    this.releaseRight();
                } else if (this.hasBothForks == 1) {
                    this.rightFork.releaseFork();
                    this.hasToEatOutside = true;
                }
            }
            if (this.allowedToEat && this.hasToEatOutside == true) {
                this.takeRight();
                this.takeLeft();
                if (this.hasBothForks == 2) {
                    this.eat();
                    this.waiter.somePhilosopherFinishedEating();
                    this.eatAttempts++;
                    this.hasToEatOutside = false;
                    this.releaseLeft();
                    this.releaseRight();
                }
            }
        }
    }
}
```

Opcje nie powodujące zakleszczenia to warianty 2, 3, 4, 5, 6 i dla nich będą przeprowadzane analizy.

Klasa do egzekwowania zadań o zadanej ilości filozofów – wyłącza wszystkie wątki po upływie określonego czasu, pobiera dane o czasach oczekiwania filozofów i printuje je oraz zapisuje w csv na podstawie której tworzyłem wykresy :

```
class Executor {  2 usages
    public long executionDuration = 20000;  1 usage

    public void executeZad1(int amount) {  no usages
        List<Fork> forks = new ArrayList<>();
        List<Philosopher> philosophers = new ArrayList<>();

        for (int i = 0; i < amount; i++) {
            forks.add(new Fork(i));
        }
        for (int i = 0; i < amount; i++) {
            Fork leftFork;
            Fork rightFork;
            if (i != amount - 1) {
                leftFork = forks.get(i);
                rightFork = forks.get(i + 1);
            } else {
                leftFork = forks.get(i);
                rightFork = forks.get(0);
            }
            Philosopher philosopher = new Philosopher(i, leftFork, rightFork);
            philosophers.add(philosopher);
            philosopher.start();
        }
        this.endExecution(philosophers);
    }
```

```java
public void executeZad2(int amount) {  no usages
    List<Fork> forks = new ArrayList<>();
    List<PhilosopherZad2> philosophers = new ArrayList<>();

    for (int i = 0; i < amount; i++) {
        forks.add(new Fork(i));
    }
    for (int i = 0; i < amount; i++) {
        Fork leftFork;
        Fork rightFork;
        if (i != amount - 1) {
            leftFork = forks.get(i);
            rightFork = forks.get(i + 1);
        } else {
            leftFork = forks.get(i);
            rightFork = forks.get(0);
        }
        PhilosopherZad2 philosopherZad2 = new PhilosopherZad2(i, leftFork, rightFork);
        philosophers.add(philosopherZad2);
        philosopherZad2.start();
    }
    endExecution(philosophers);
}
```

```java
public void executeZad3(int amount) {  no usages
    List<Fork> forks = new ArrayList<>();
    List<PhilosopherZad3> philosophers = new ArrayList<>();

    for (int i = 0; i < amount; i++) {
        forks.add(new Fork(i));
    }
    for (int i = 0; i < amount; i++) {
        Fork leftFork;
        Fork rightFork;
        if (i != amount - 1) {
            leftFork = forks.get(i);
            rightFork = forks.get(i + 1);
        } else {
            leftFork = forks.get(i);
            rightFork = forks.get(0);
        }
        PhilosopherZad3 philosopherZad3 = new PhilosopherZad3(i, leftFork, rightFork);
        philosophers.add(philosopherZad3);
        philosopherZad3.start();
    }
    endExecution(philosophers);
}
```

```java
public void executeZad4(int amount) {  no usages
    List<Fork> forks = new ArrayList<>();
    List<PhilosopherZad4> philosophers = new ArrayList<>();

    for (int i = 0; i < amount; i++) {
        forks.add(new Fork(i));
    }
    for (int i = 0; i < amount; i++) {
        Fork leftFork;
        Fork rightFork;
        if (i != amount - 1) {
            leftFork = forks.get(i);
            rightFork = forks.get(i + 1);
        } else {
            leftFork = forks.get(i);
            rightFork = forks.get(0);
        }
        PhilosopherZad4 philosopherZad4 = new PhilosopherZad4(i, leftFork, rightFork);
        philosophers.add(philosopherZad4);
        philosopherZad4.start();
    }
    endExecution(philosophers);
}
```

```java
public void executeZad5(int amount) {  no usages
    List<Fork> forks = new ArrayList<>();
    List<PhilosopherZad5> philosopherList = new ArrayList<>();
    Waiter waiter = new Waiter();

    for (int i = 0; i < amount; i++) {
        forks.add(new Fork(i));
    }
    for (int i = 0; i < amount; i++) {
        Fork leftFork;
        Fork rightFork;
        if (i != amount - 1) {
            leftFork = forks.get(i);
            rightFork = forks.get(i + 1);
        } else {
            leftFork = forks.get(i);
            rightFork = forks.get(0);
        }
        PhilosopherZad5 philosopherZad5 = new PhilosopherZad5(i, leftFork, rightFork, waiter);
        philosopherList.add(philosopherZad5);
        Thread currThread = new Thread(philosopherZad5);
        waiter.addPhilosopher(philosopherZad5, currThread);

    }
    waiter.start();
    for (PhilosopherZad5 philosopher : philosopherList) {
        philosopher.start();
    }
    endExecution(philosopherList);
}
```

```java
public void executeZad6(int amount) {  no usages
    List<Fork> forks = new ArrayList<>();
    List<PhilosopherZad6> philosopherList = new ArrayList<>();
    WaiterZad6 waiter = new WaiterZad6();

    for (int i = 0; i < amount; i++) {
        forks.add(new Fork(i));
    }
    for (int i = 0; i < amount; i++) {
        Fork leftFork;
        Fork rightFork;
        if (i != amount - 1) {
            leftFork = forks.get(i);
            rightFork = forks.get(i + 1);
        } else {
            leftFork = forks.get(i);
            rightFork = forks.get(0);
        }
        PhilosopherZad6 philosopherZad6 = new PhilosopherZad6(i, leftFork, rightFork, waiter);
        philosopherList.add(philosopherZad6);
        Thread currThread = new Thread(philosopherZad6);
        waiter.addPhilosopher(philosopherZad6, currThread);

    }
    waiter.start();
    for (PhilosopherZad6 philosopher : philosopherList) {
        philosopher.start();
    }
    endExecution(philosopherList);
}
```

```java
private void endExecution(List<? extends Thread> philosophers) {  6 usages
    try {
        Thread.sleep(executionDuration);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }


    for (Thread philosopher : philosophers) {
        philosopher.interrupt();
    }


    Map<Integer, Double> averageWaitTimes = new HashMap<>();
    for (Thread philosopher : philosophers) {
        if (philosopher instanceof Philosopher) {
            System.out.println("First Philosopher can't be measured because there can be deadlock");
        } else if (philosopher instanceof PhilosopherZad2) {
            PhilosopherZad2 p = (PhilosopherZad2) philosopher;
            averageWaitTimes.put(p.number, p.getAverageWaitTime());
        } else if (philosopher instanceof PhilosopherZad3) {
            PhilosopherZad3 p = (PhilosopherZad3) philosopher;
            averageWaitTimes.put(p.number, p.getAverageWaitTime());
        } else if (philosopher instanceof PhilosopherZad4) {
            PhilosopherZad4 p = (PhilosopherZad4) philosopher;
            averageWaitTimes.put(p.number, p.getAverageWaitTime());
        } else if (philosopher instanceof PhilosopherZad5) {
            PhilosopherZad5 p = (PhilosopherZad5) philosopher;
            averageWaitTimes.put(p.number, p.getAverageWaitTime());
```

```java
private void endExecution(List<? extends Thread> philosophers) {  6 usages
            PhilosopherZad6 p = (PhilosopherZad6) philosopher;
            averageWaitTimes.put(p.number, p.getAverageWaitTime());
        }
    }

    this.printAverageTimes(averageWaitTimes);
    this.writeCSV(averageWaitTimes);
}

private void printAverageTimes(Map<Integer, Double> averageWaitTimes) {  1 usage
    System.out.println("ID Filozofa\tŚredni czas oczekiwania (ms)");
    for (Entry<Integer, Double> entry : averageWaitTimes.entrySet()) {
        System.out.println("Filozof " + entry.getKey() + "\t" + entry.getValue());
    }
}
private void writeCSV(Map<Integer, Double> averageWaitTimes) {  1 usage
    try (BufferedWriter writer = new BufferedWriter(new FileWriter( fileName: "average_wait_times.csv"))) {
        writer.write( str: "ID Filozofa, Średni Czas Oczekiwania (ms)\n");


        for (Entry<Integer, Double> entry : averageWaitTimes.entrySet()) {
            writer.write( str: entry.getKey() + "," + entry.getValue() + "\n");
        }

        System.out.println("Zapisano do average_wait_times.csv");
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Main gdzie wszystko jest wywoływane :

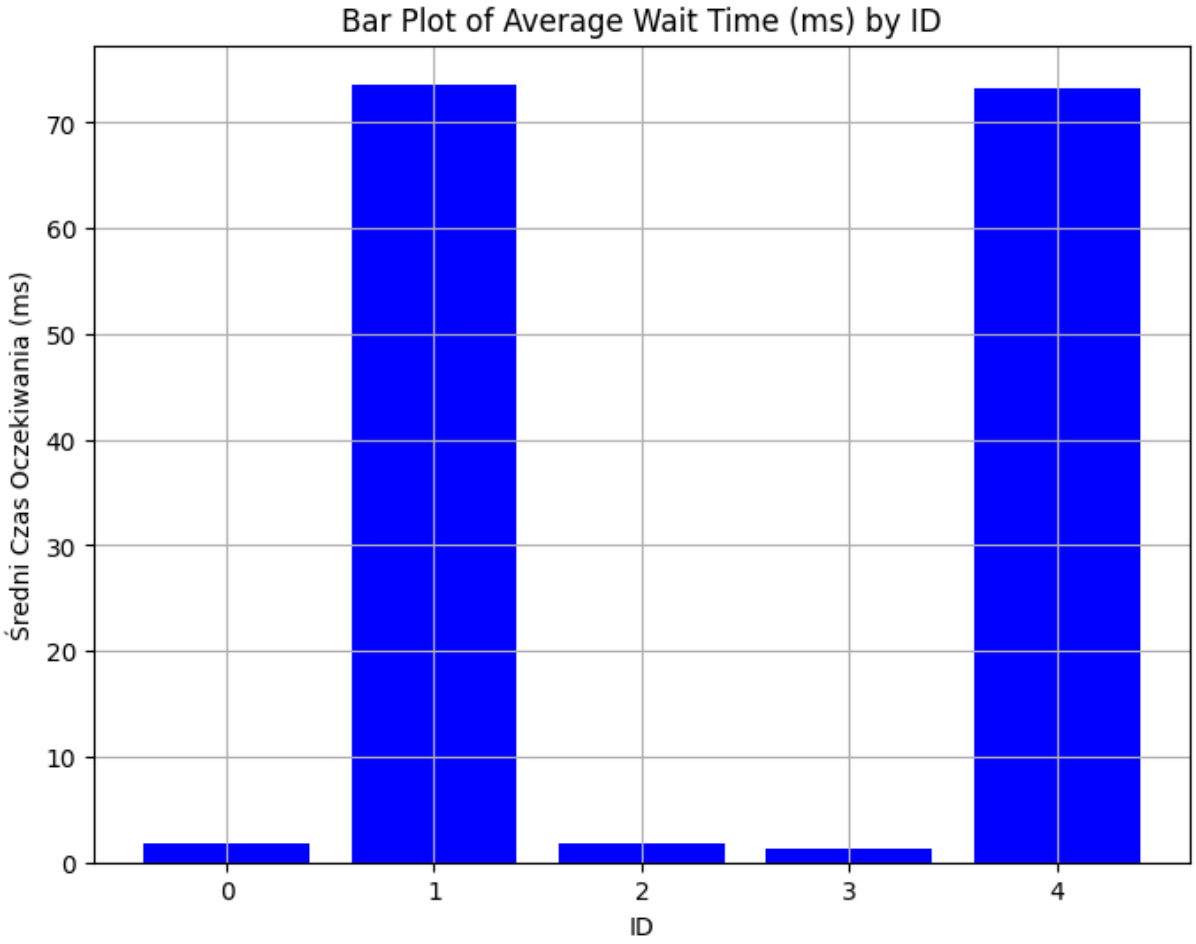
```java
public class Main {
    public static void main(String[] args) {
        int firstPhilosophersAmount = 5;
        int secondPhilosophersAmount = 10;
        int thirdPhilosophersAmount = 20;
        Executor executor = new Executor();

        //executor.executeZad1(firstPhilosophersAmount);
        //executor.executeZad1(secondPhilosophersAmount);
        //executor.executeZad1(thirdPhilosophersAmount);

        //executor.executeZad2(firstPhilosophersAmount);
        //executor.executeZad2(secondPhilosophersAmount);
        //executor.executeZad2(thirdPhilosophersAmount);

        //executor.executeZad3(firstPhilosophersAmount);
        //executor.executeZad3(secondPhilosophersAmount);
        //executor.executeZad3(thirdPhilosophersAmount);

        //executor.executeZad4(firstPhilosophersAmount);
        //executor.executeZad4(secondPhilosophersAmount);
        //executor.executeZad4(thirdPhilosophersAmount);

        //executor.executeZad5(firstPhilosophersAmount);
        //executor.executeZad5(secondPhilosophersAmount);
        //executor.executeZad5(thirdPhilosophersAmount);

        //executor.executeZad6(firstPhilosophersAmount);
        //executor.executeZad6(secondPhilosophersAmount);
        //executor.executeZad6(thirdPhilosophersAmount);
    }
```

Statystyki

Zadanie 2 – 5 filozofów :

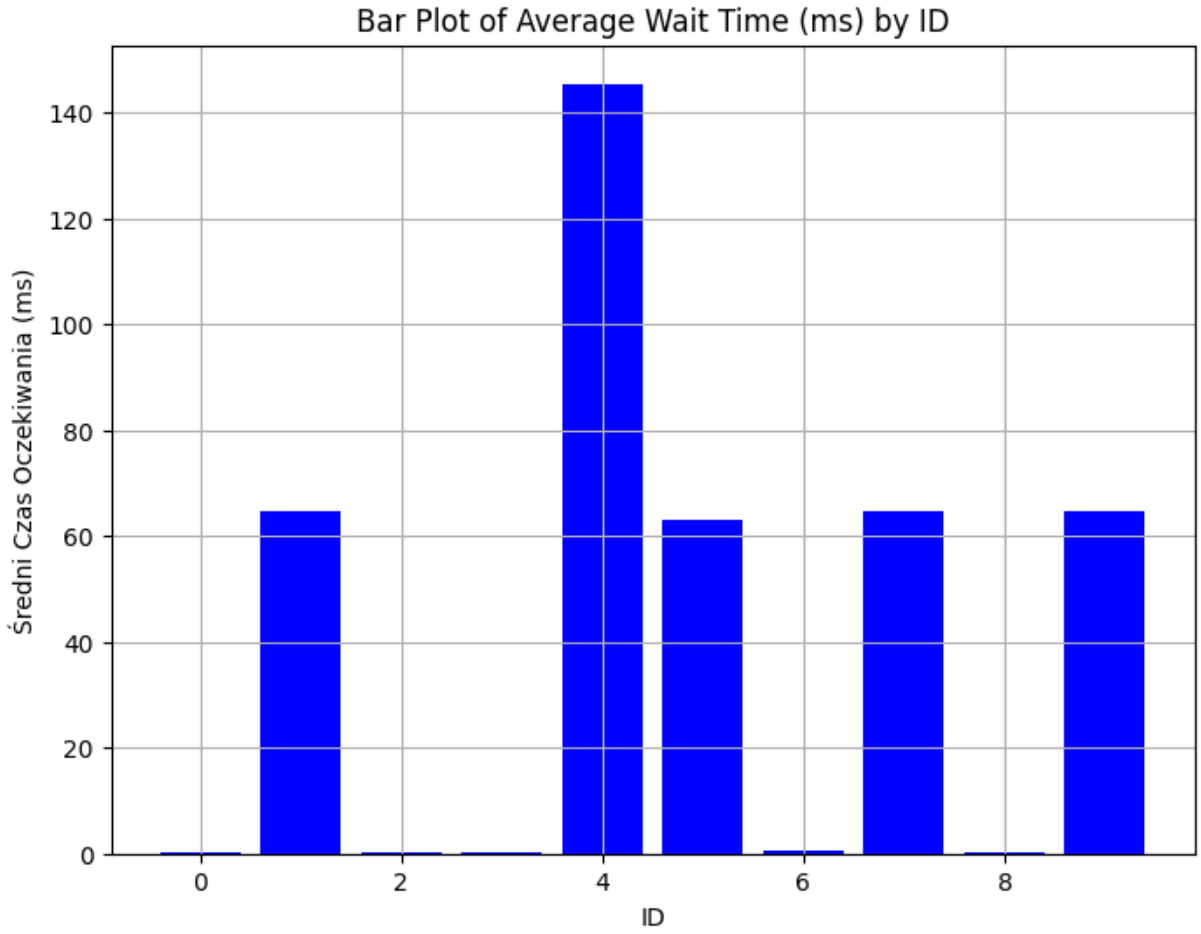
1 pomiar :

ID, Average wait time (ms)
0,1.778
1,73.62647142857143
2,1.8163857142857143
3,1.267825
4,73.2882



Zadanie 2 – 10 filozofów

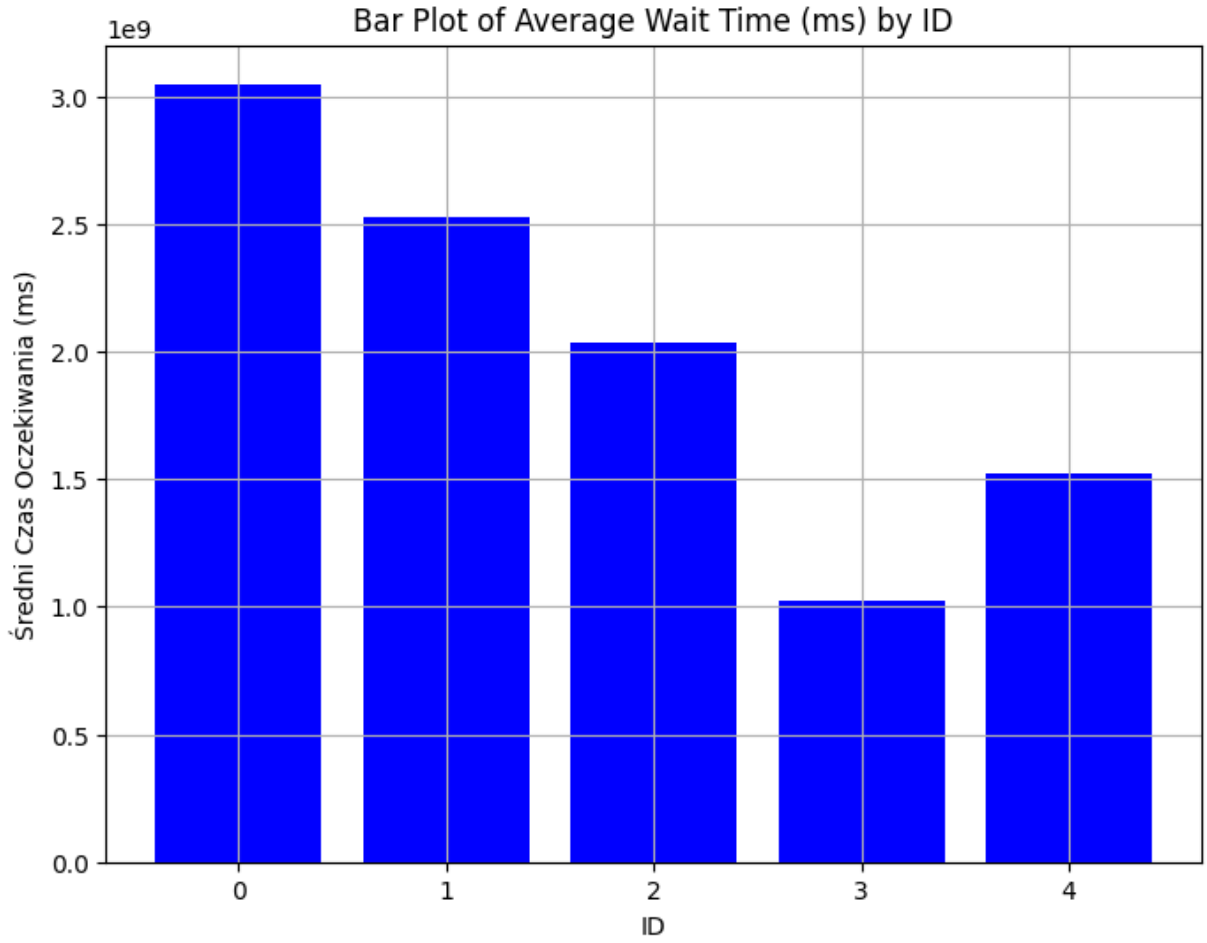
ID, Average wait time (ms)
0,0.2969
1,64.5721875
2,0.3487
3,0.1293
4,145.4697
5,62.91585
6,0.5658125
7,64.547325
8,0.3095625
9,64.7669125



Zadanie 3 – 5 filozofów :

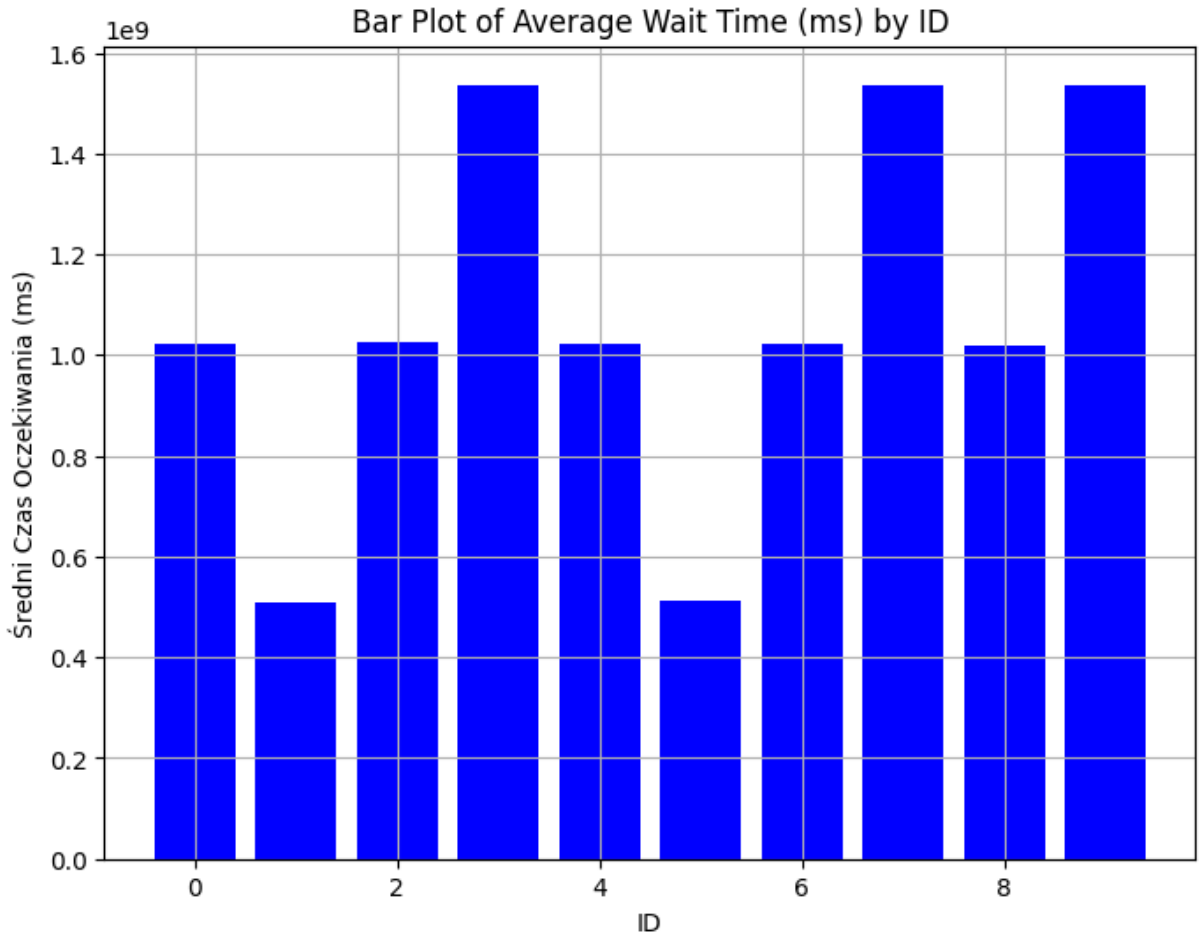
ID, Average wait time (ms)
0,3.0477353E9
1,2.527722E9

2,2.0363098E9
3,1.0232746E9
4,1.5186494E9



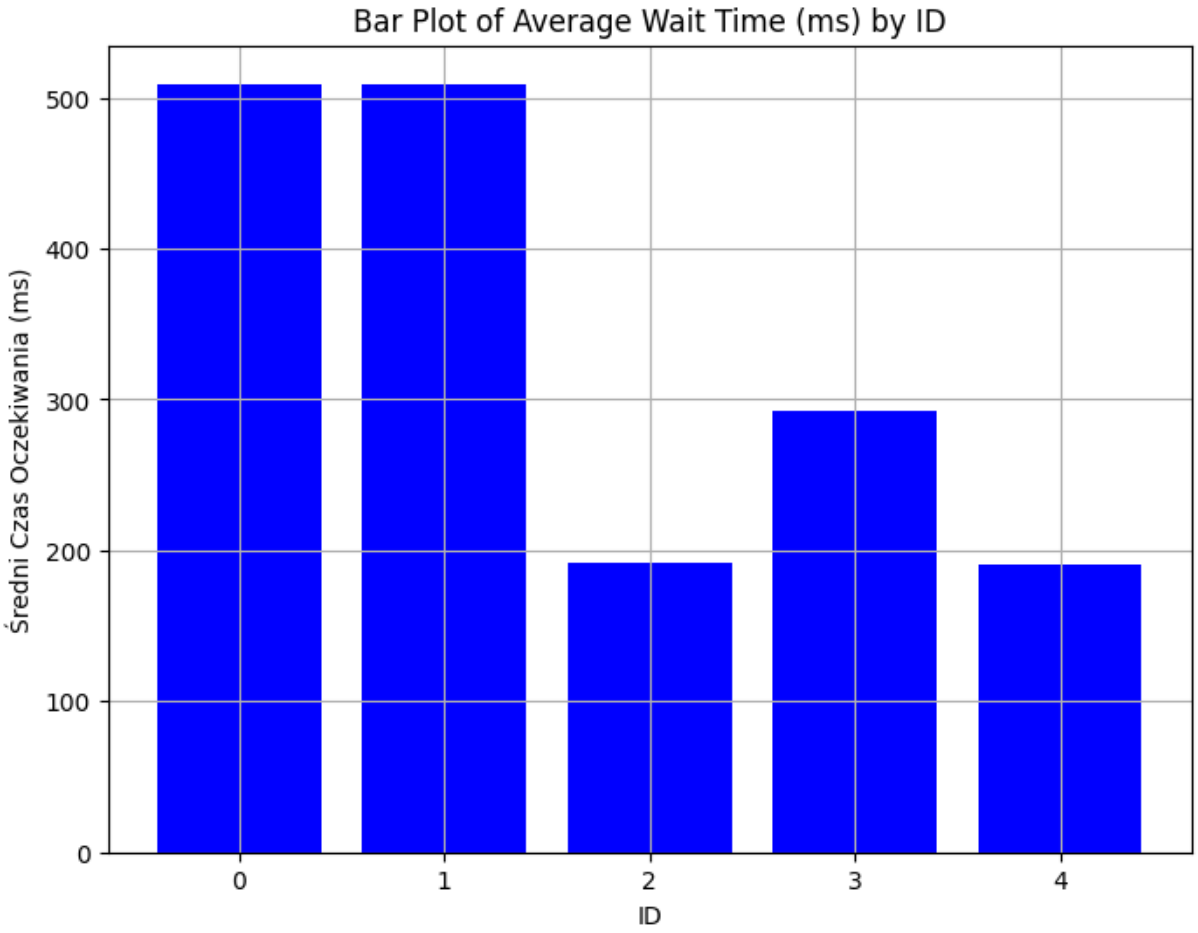
Zadanie 3 – 10 filozofów :

ID, Average wait time (ms)
0,1.0231442E9
1,5.094715E8
2,1.0241495E9
3,1.5366062E9
4,1.0231871E9
5,5.113106E8
6,1.0238573E9
7,1.535421E9
8,1.0192946E9
9,1.5365747E9



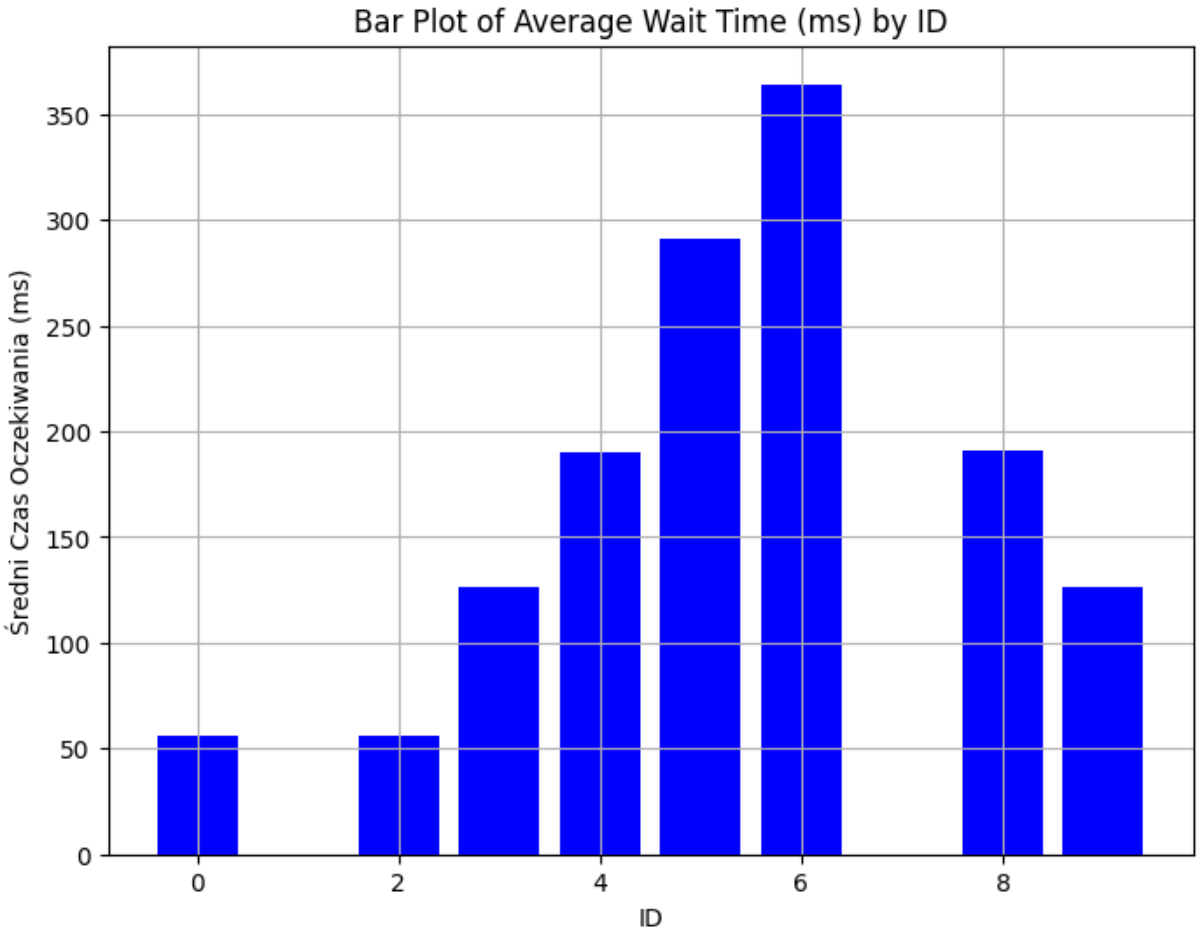
Zadanie 4 – 5 filozofów :

ID, Average wait time (ms)
0,508.96066666666667
1,509.16965
2,191.3626875
3,292.9872714285714
4,190.82385



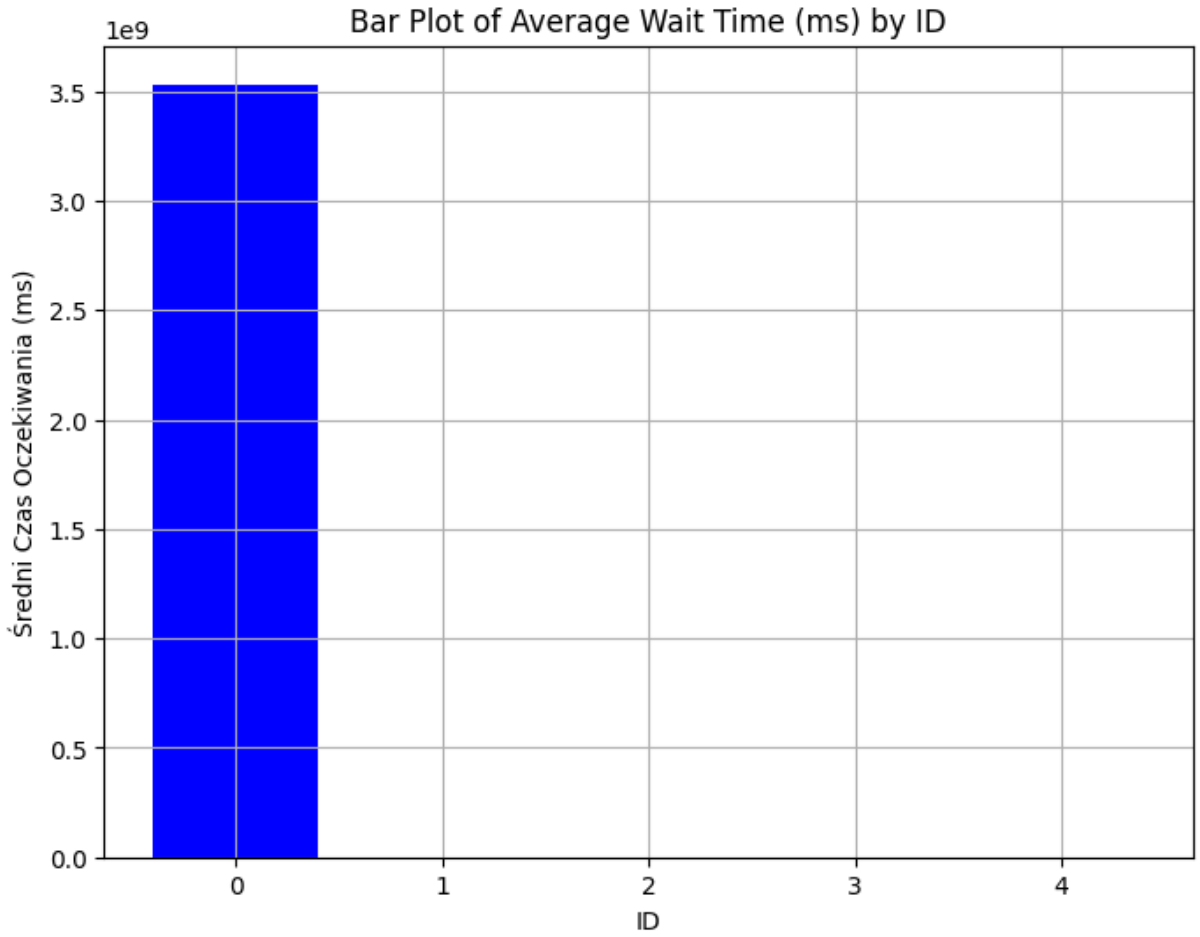
Zadanie 4 – 10 filozofów :

ID, Average wait time (ms)
0,55.89231111111111
1,0.0042
2,55.92428888888889
3,126.61675
4,190.461625
5,290.7702714285714
6,364.02288571428574
7,0.0016666666666666668
8,190.5925625
9,126.5961875



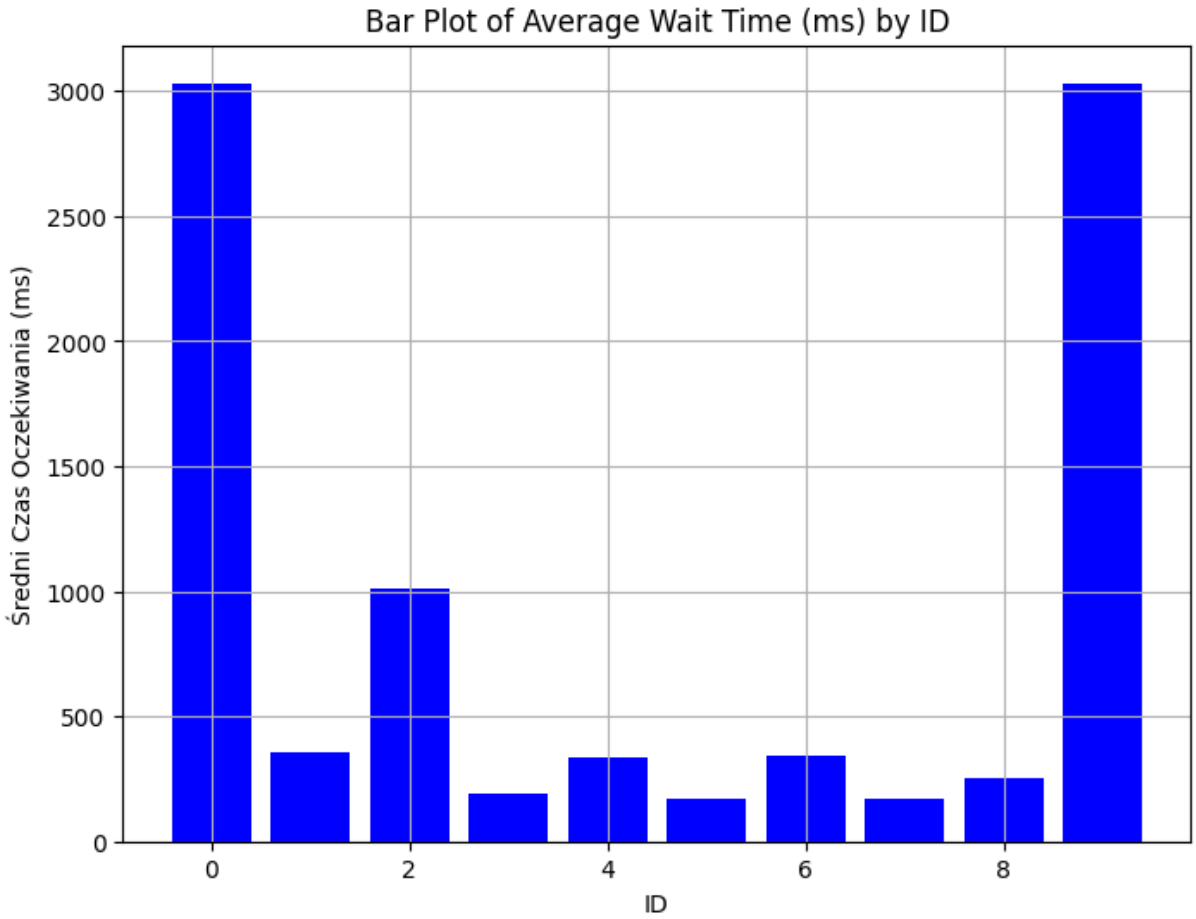
Zadanie 5 – 5 filozofów

ID, Average wait time (ms)
0,3.5328966E9
1,2264.48335
2,404.00734
3,188.650175
4,0.41635



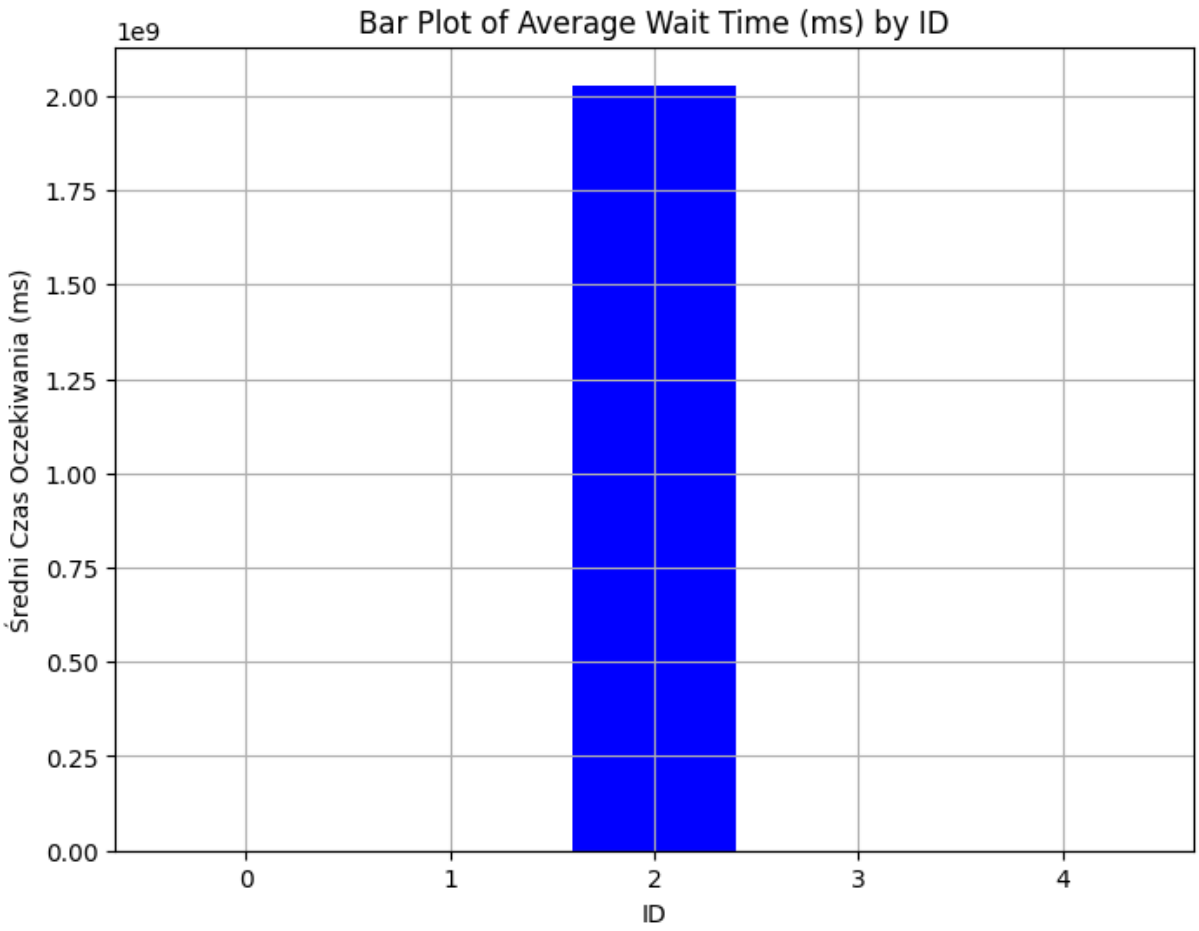
Zadanie 5 – 10 filozofów :

ID, Average wait time (ms)
0,3029.3754
1,359.3984571428571
2,1013.34585
3,189.9962625
4,335.44083333333333
5,167.379
6,340.4108
7,167.88303333333334
8,252.3886
9,3031.0949



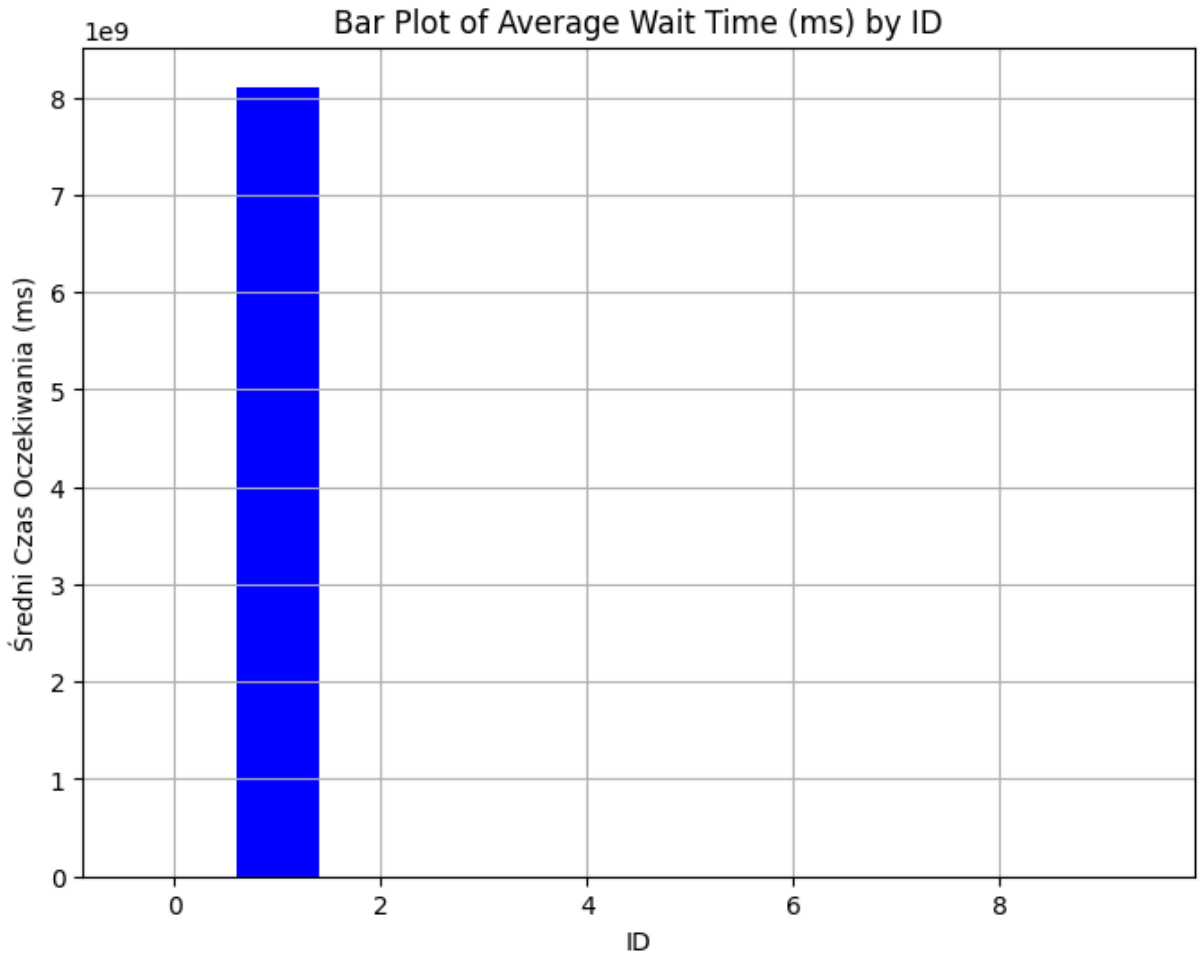
Zadanie 6 – 5 filozofów :

ID, Average wait time (ms)
0,127.215575
1,0.2724
2,2.0280507E9
3,1525.6617
4,190.40405



Zadanie 6 – 10 filozofów :

ID, Average wait time (ms)
0,3.10972
1,8.1093588E9
2,8112.4257
3,592.5946666666666
4,607.94706
5,510.63644
6,405.45468
7,307.84392
8,201.90942
9,104.12532



Wnioski

Jak widzimy, warianty bardzo się od siebie różnią. W dwóch ostatnich, gdzie zagłodzenie jest bardzo częste, jeden filozof znacznie dominuje w czasie oczekiwania na widelce, aczkolwiek nie jest to aż tak bardzo złe bo tylko jeden głoduje a reszta jest w miarę porównywalna. W Drugim przykładzie więcej filozofów głoduje, ale robi to krótszy czas a przypadki 3 i 4 wydają się być niezłym kompromisem co do zagładzania bo filozofowie są w miarę wyrównani czasowo.

Średnie czasy dla wariantów z możliwością zagłodzenia są wyższe i bardziej skumulowane wokół poszczególnych filozofów.

Brak mechanizmów synchronizacji może zdecydowanie zwiększyć czas oczekiwania na dostęp do zasobów. Gdyby nie było synchronized, więcej wątków mogłoby utknąć w jakiejś sekcji i wyjść z niej dużo później niż byliby w stanie z użyciem tego mechanizmu.