Laboratorium 4

Problem pięciu filozofów

Franciszek Jawor

1. Wstęp

Problem pięciu filozofów jest klasycznym problemem programowania współbieżnego. W tym ćwiczeniu przedstawię 4 proponowane rozwiązania i porównam ich efektywność.

2. Rozwiązania problemu

Poniższe klasy abstrakcyjne są podstawą każdego rozwiązania:

```
public abstract class BasePhilosopher extends Thread
{
    protected final int id;
    protected final int leftFork;
    protected final int rightFork;
    protected int eatingCount = 0;
    protected List<Long> waitingTimes = new ArrayList<Long>();

    public BasePhilosopher(int id, int n)
    {
        this.id = id;
        leftFork = id;
        rightFork = (id + 1) % n;
    }

    public int GetEatingCount()
    {
        return eatingCount;
    }

    public List<Long> GetWaitingTimes()
    {
        return new ArrayList<>(waitingTimes);
    }

    public int GetId()
    {
        return id;
    }
}
```

```
public class AbstractTable
{
    protected boolean[] forks;

    public AbstractTable(int forkCount)
    {
        forks = new boolean[forkCount];

        for(int i = 0; i < forkCount; i++)
            forks[i] = true;
    }
}
```

Rozwiązanie naiwne

W tym rozwiązaniu każdy z filozofów próbuje wziąć lewą pałeczkę, a następnie próbuje wziąć prawą i w przypadku niepowodzenia- czeka na zwolnienie zasobu. W sytuacji, w której każdy filozof weźmie po jednej pałeczce, następuje zakleszczenie i nikt więcej nie zje. Z tego względu to rozwiązanie nie jest poprawne.

```
public class NaivePhilosopher extends BasePhilosopher
{
    private final NaiveTable _table;
    private final int _id;

    public NaivePhilosopher(NaiveTable table, int id, int n)
    {
        super(id, n);
        _table = table;
        _id = id;
    }

    public void run()
    {
        var random = new Random();

        try
        {
            while (true)
            {
                System.out.println("Philosopher " + _id + " is THINKING");
                Thread.sleep(random.nextInt(100));

                long startTime = System.currentTimeMillis();
                _table.TakeFork(leftFork);
                System.out.println("Philosopher " + _id + " took left fork " +
leftFork);

                _table.TakeFork(rightFork);
                long endTime = System.currentTimeMillis();
                System.out.println("Philosopher " + _id + " took right fork " +
rightFork);

                eatingCount++;
                waitingTimes.add(endTime - startTime);

                System.out.println("Philosopher " + _id + " is EATING");
                Thread.sleep(random.nextInt(100));

                _table.PlaceForkBack(leftFork);
                _table.PlaceForkBack(rightFork);
                System.out.println("Philosopher " + _id + " put forks back");
            }
        }
        catch (InterruptedException e)
        {
            Thread.currentThread().interrupt();
        }
    }
}
```

```
public class NaiveTable extends AbstractTable
{
    public NaiveTable(int forkCount)
    {
        super(forkCount);
    }

    public synchronized void TakeFork(int forkId) throws InterruptedException
    {
        while(!forks[forkId])
            wait();

        forks[forkId] = false;
    }

    public synchronized void PlaceForkBack(int forkId)
    {
        forks[forkId] = true;
        notifyAll();
    }
}
```

Rozwiązanie z zagłodzeniem

W tym przypadku każdy filozof próbuje od razu wziąć obie pałeczki i czeka jeśli mu się to nie uda. Dzięki temu unikamy zakleszczenia, gdyż zawsze ktoś będzie jadł, jednak odbywa się to kosztem możliwości zagłodzenia. Gdy dwóch sąsiadów będzie jadło naprzemiennie, jeden z filozofów nie będzie miał możliwości zjedzenia.

```
public class StarvingPhilosopher extends BasePhilosopher
{
    private final StarvingTable _table;
    private final int _id;

    public StarvingPhilosopher(StarvingTable table, int id, int n)
    {
        super(id, n);
        _table = table;
        _id = id;
    }

    public void run()
    {
        var random = new Random();

        try
        {
            while (true)
            {
                System.out.println("Philosopher " + _id + " is THINKING");
                Thread.sleep(random.nextInt(100));

                long startTime = System.currentTimeMillis();
                _table.TakeBothForks(leftFork, rightFork);
                long endTime = System.currentTimeMillis();
                System.out.println("Philosopher " + _id + " took both forks");

                eatingCount++;
                waitingTimes.add(endTime - startTime);

                System.out.println("Philosopher " + _id + " is EATING");
                Thread.sleep(random.nextInt(100));

                _table.PlaceForksBack(leftFork, rightFork);
                System.out.println("Philosopher " + _id + " put forks back");
            }
        }
        catch (InterruptedException e)
        {
            Thread.currentThread().interrupt();
        }
    }
}
```

```
public class StarvingTable extends AbstractTable
{
    public StarvingTable(int forkCount)
    {
        super(forkCount);
    }

    public synchronized void TakeBothForks(int leftIndex, int rightIndex) throws
InterruptedException
    {
        if(!forks[leftIndex] || !forks[rightIndex])
            wait();

        forks[leftIndex] = false;
        forks[rightIndex] = false;
    }

    public synchronized void PlaceForksBack(int leftIndex, int rightIndex)
    {
        forks[leftIndex] = true;
        forks[rightIndex] = true;

        notifyAll();
    }
}
```

Rozwiązanie asymetryczne

W tym rozwiąniu numerujemy filozofów. Następnie filozofowie z parzystymi numerami biorą lewe pałeczki, a z nieparzystymi- prawe. Dzięki temu eliminujemy możliwość zakleszczenia. Natomiast zagłodzenie jest teoretycznie możliwe, ale wydaje mi się, że wystąpi dużo rzadziej niż w poprzednim rozwiązaniu.

```
public class EvenOddPhilosopher extends BasePhilosopher
{
    private final NaiveTable _table;
    private final int _id;

    public EvenOddPhilosopher(NaiveTable table, int id, int n)
    {
        super(id, n);
        _table = table;
        _id = id;
    }

    public void run()
    {
        var random = new Random();

        try
        {
            while (true)
            {
                System.out.println("Philosopher " + _id + " is THINKING");
                Thread.sleep(random.nextInt(100));

                long startTime = System.currentTimeMillis();
                if(_id % 2 == 0)
                {
                    _table.TakeFork(leftFork);
                    _table.TakeFork(rightFork);
                }
                else
                {
                    _table.TakeFork(rightFork);
                    _table.TakeFork(leftFork);
                }
                long endTime = System.currentTimeMillis();
                eatingCount++;
                waitingTimes.add(endTime - startTime);

                System.out.println("Philosopher " + _id + " is EATING");
                Thread.sleep(random.nextInt(100));

                _table.PlaceForkBack(leftFork);
                _table.PlaceForkBack(rightFork);
                System.out.println("Philosopher " + _id + " put forks back");
            }
        }
        catch (InterruptedException e)
        {
            Thread.currentThread().interrupt();
        }
    }
}
```

Rozwiązanie z arbitrem

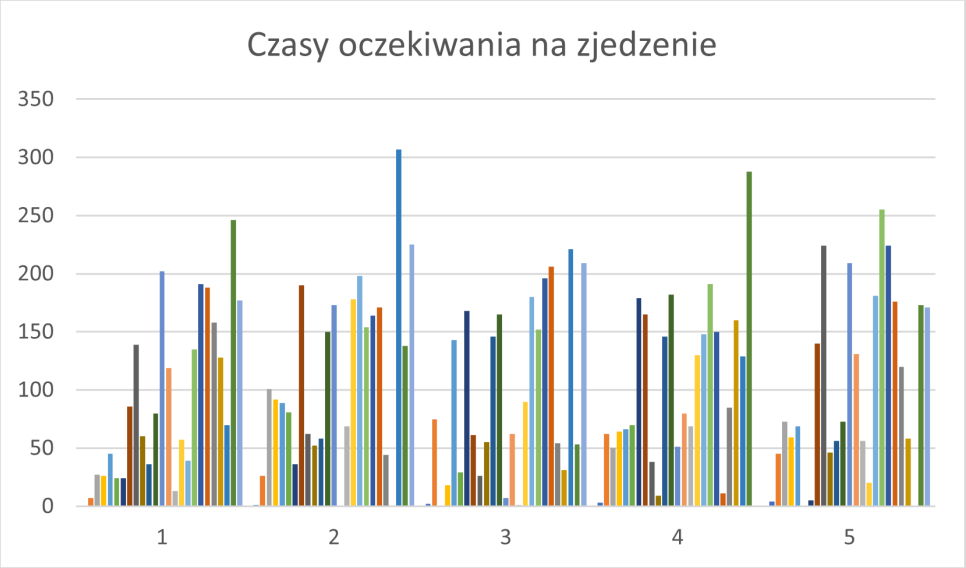
W tym rozwiązaniu potrzebny jest dodatkowy mechanizm arbitra, który będzie pilnował, żeby co najmniej jeden filozof nie konkurował o dostęp do pałeczek.

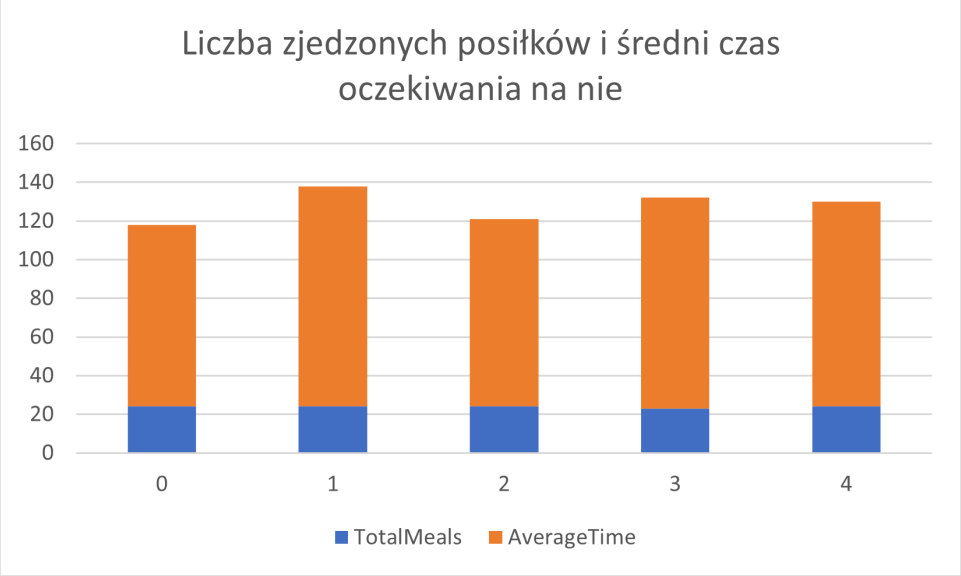
```
public class ArbiterPhilosopher extends BasePhilosopher
{
    private final ArbiterTable _table;
    private final int _id;

    public ArbiterPhilosopher(ArbiterTable table, int id, int n)
    {
        super(id, n);
        _table = table;
        _id = id;
    }


    public void run()
    {
        var random = new Random();

        try
        {
            while (true)
            {
                System.out.println("Philosopher " + _id + " is THINKING");
                Thread.sleep(random.nextInt(100));

                long startTime = System.currentTimeMillis();
                _table.TryTakingBothForks(leftFork, rightFork);
                long endTime = System.currentTimeMillis();
                System.out.println("Philosopher " + _id + " took both forks");

                eatingCount++;
                waitingTimes.add(endTime - startTime);

                System.out.println("Philosopher " + _id + " is EATING");
                Thread.sleep(random.nextInt(100));

                _table.PlaceForksBack(leftFork, rightFork);
                System.out.println("Philosopher " + _id + " put forks back");
            }
        }
        catch (InterruptedException e)
        {
            Thread.currentThread().interrupt();
            System.out.println("Philosopher " + _id + " was interrupted");
        }
    }
}
```

```
public class ArbiterTable extends AbstractTable
{
    private final int MAX_EATING_AT_ONCE;

    private int _currentlyEating = 0;

    public ArbiterTable(int philosophersCount)
    {
        super(philosophersCount);

        MAX_EATING_AT_ONCE = philosophersCount - 1;
    }

    public synchronized void TryTakingBothForks(int leftIndex, int rightIndex)
throws InterruptedException {
        while (_currentlyEating >= MAX_EATING_AT_ONCE || !forks[leftIndex] || !
forks[rightIndex])
        {
            wait();
        }

        forks[leftIndex] = false;
        forks[rightIndex] = false;
        _currentlyEating++;
    }

    public synchronized void PlaceForksBack(int leftIndex, int rightIndex)
    {
        forks[leftIndex] = true;
        forks[rightIndex] = true;
        _currentlyEating--;

        notifyAll();
    }
}
```

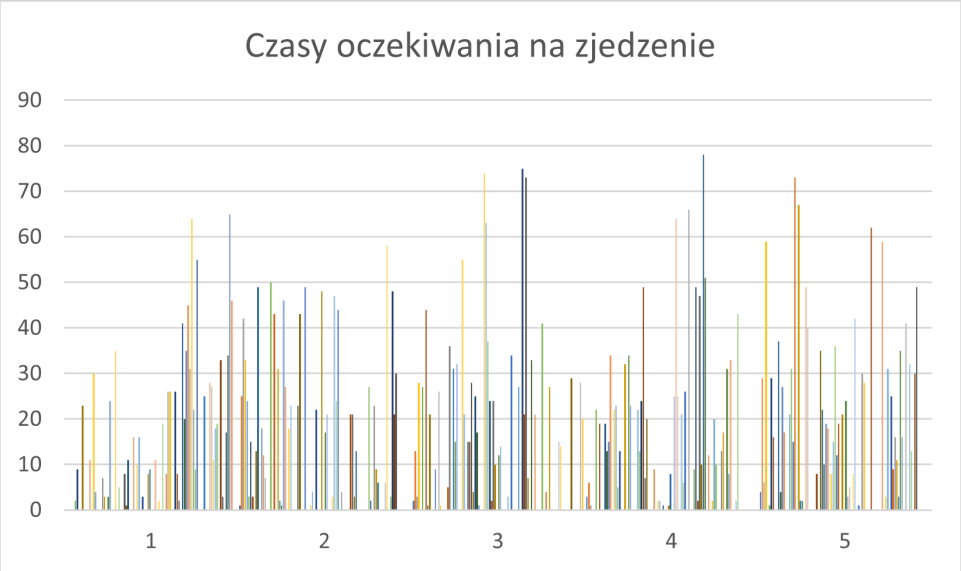
3. Wyniki

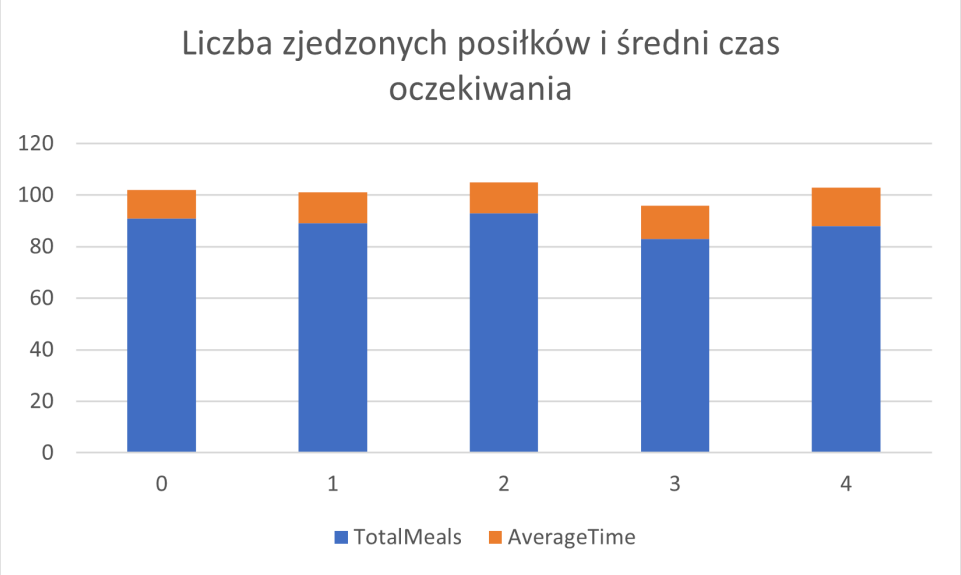
Rozwiązanie naiwne



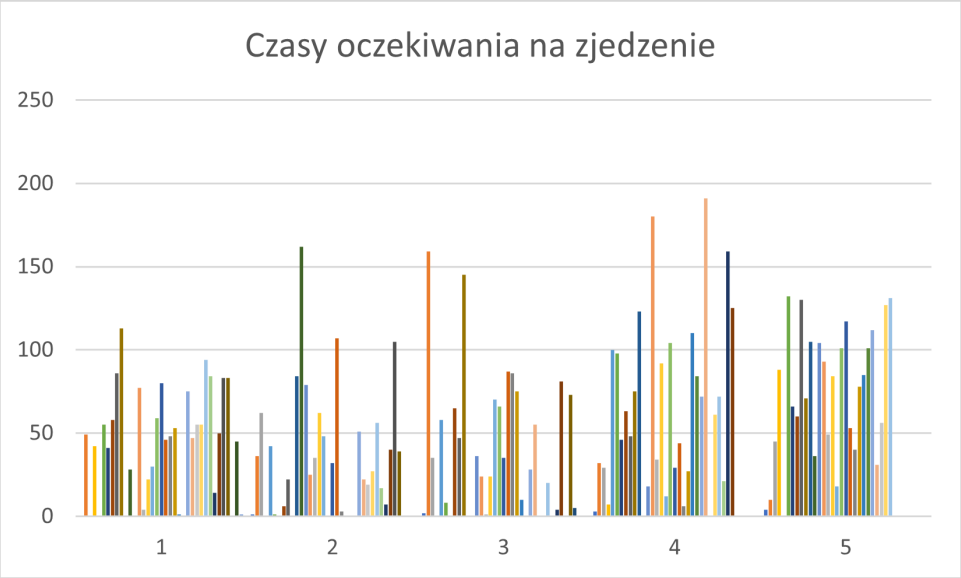


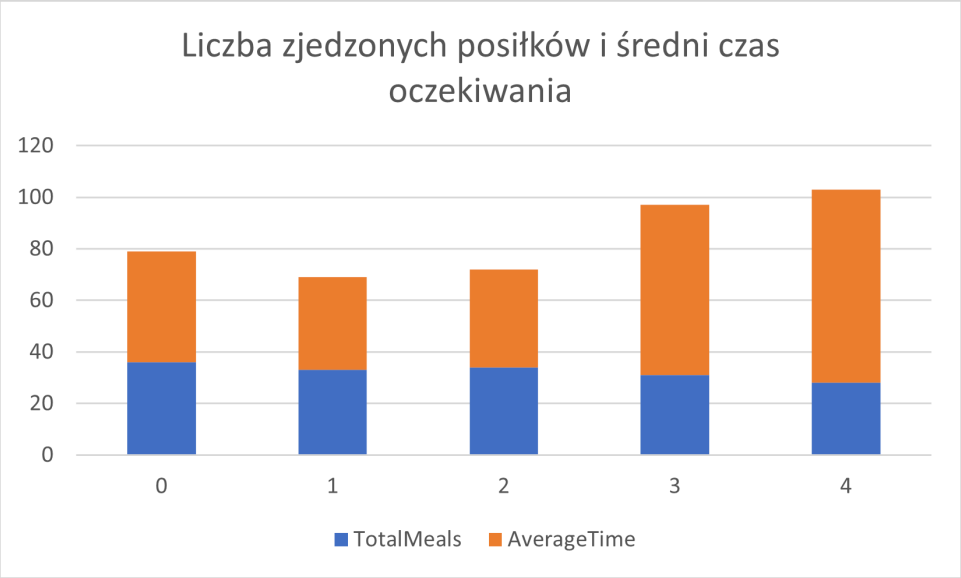
Rozwiązanie z zagłodzeniem



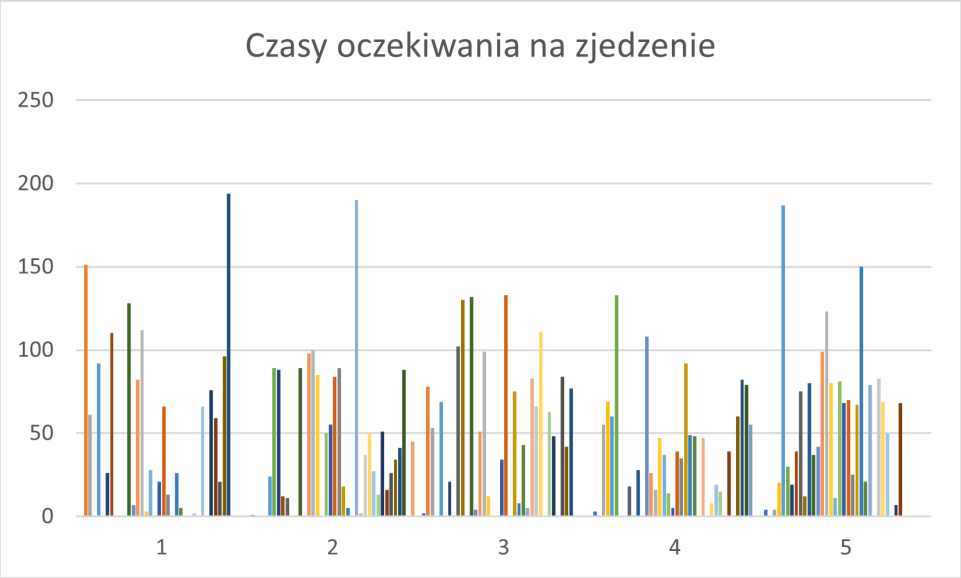


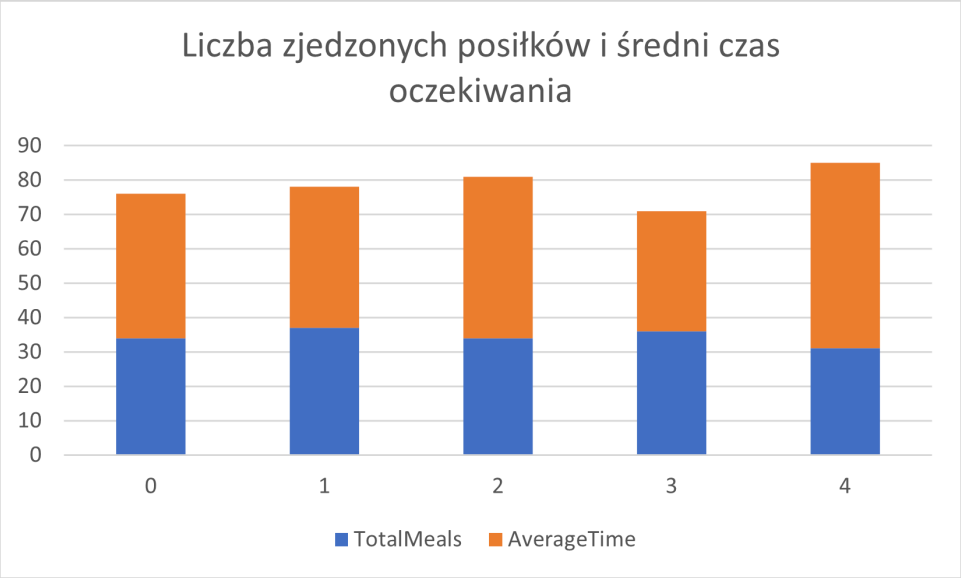
Rozwiązanie asymetryczne





Rozwiązanie z arbitrem





4.Wnioski

Najlepsze pod kątem wydajności (mimo, że niepoprawne) okazało się być rozwiązanie z zagłodzeniem. Pomimo możliwości zagłodzenia każdy z filozofów był w stanie zjeść prawie dwa razy więcej posiłków, niż w innych rozwiązaniach. Rozwiązanie z arbitrem i asymetryczne okazały się bardzo podobne pod kątem wydajności i liczby posiłków. Natomiast rozwiązanie naiwne, jak można się było spodziewać, miało najgorsze wyniki. Jak widać projektowanie wydajnych rozwiązań problemów programowania współbieżnego, nie jest trywialne, a w dodatku zrobienie tego poprawnie, jest jeszcze trudniejsze.