# Laboratorium 5

**Łukasz Wala**
*AGH, Wydział Informatyki, Elektroniki i Telekomunikacji*
*Teoria Współbieżności 2022/23*

Kraków, 16 listopada 2022

## 1 Treść zadania

Problem pięciu filozofów:

1. Zaimplementować trywialne rozwiązanie z symetrycznymi filozofami. Zaobserwować problem blokady.
2. Zaimplementować rozwiązanie z widelcami podnoszonymi jednocześnie. Jaki problem może tutaj wystąpić?
3. Zaimplementować rozwiązanie z lokajem.
4. Wykonać pomiary dla każdego rozwiązania i wywnioskować co ma wpływ na wydajność każdego rozwiązania.

## 2 Rozwiązanie trywialne

Poniżej implementacja problemu:

```
class Fork {
    private boolean available = true;

    public synchronized void pickUp() {
        while (!available) {
            try {
                wait();
            } catch (InterruptedException e) {
                System.exit(0);
            }
        }

        available = false;
    }
```

```
    public synchronized void putDown() {
        available = true;
        notifyAll();
    }
}

class Philosopher extends Thread {
    private int counter = 0;
    private Fork rightFork;
    private Fork leftFork;

    public Philosopher(Fork leftFork, Fork rightFork) {
        this.leftFork = leftFork;
        this.rightFork = rightFork;
    }

    public void run() {
        while (true) {

        thinkOrEat(500);

        leftFork.pickUp();
        rightFork.pickUp();

        thinkOrEat(100);
        ++counter;
        if (counter % 100 == 0) {
            System.out.println("Philosopher: " + Thread.currentThread() +
                " have eaten " + counter + " times");
        }

        leftFork.putDown();
        rightFork.putDown();
        }
    }

    public void thinkOrEat(long time) {
        try {
            Thread.sleep(time);
        } catch (InterruptedException e) {
            System.exit(0);
        }
    }
}
```

```
public class Fil5mon {
    public static void main(String[] args) throws InterruptedException {
        int n = 5;

        Philosopher[] philosophers = new Philosopher[n];
        Fork[] forks = new Fork[n];

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

        for (int i=0; i<n; ++i) {
            philosophers[i] = new Philosopher(forks[i], forks[(i+1)%n]);
        }

        for (int i=0; i<n; ++i) {
            philosophers[i].start();
        }

        for (int i=0; i<n; ++i) {
            philosophers[i].join();
        }
    }
}
```

Problem w tym rozwiązaniu występuje w sytuacji, gdy wszyscy filozofowie podniosą np. swój lewy widelec. Wówczas każdy z nich będzie czekał na swój prawy widelec w nieskończoność.

# 3 Rozwiązanie z widelcami podnoszonymi jednocześnie

```
class Fork {
    private boolean available = true;

    public synchronized void pickUp() {
        while (!available) {
            try {
                wait();
            } catch (InterruptedException e) {
                System.exit(0);
            }
        }

        available = false;
```

```
    }

    public synchronized void putDown() {
        available = true;
        notifyAll();
    }
}

class Philosopher extends Thread {
    private int counter = 0;
    private Fork rightFork;
    private Fork leftFork;
    private Object lock;

    public Philosopher(Fork leftFork, Fork rightFork, Object lock) {
        this.leftFork = leftFork;
        this.rightFork = rightFork;
        this.lock = lock;
    }

    public void run() {
        while (true) {

        thinkOrEat(500);

        synchronized(lock) {
            leftFork.pickUp();
            rightFork.pickUp();
        }

        thinkOrEat(100);
        ++counter;
        if (counter % 100 == 0) {
            System.out.println("Philosopher: " + Thread.currentThread() +
              " have eaten " + counter + " times");
        }

        leftFork.putDown();
        rightFork.putDown();
      }
    }

    public void thinkOrEat(long time) {
        try {
            Thread.sleep(time);
        } catch (InterruptedException e) {
```

```
            System.exit(0);
        }
    }
}

public class Fil5mon {
    public static void main(String[] args) throws InterruptedException {
        int n = 5;
        Object lock = new Object();

        Philosopher[] philosophers = new Philosopher[n];
        Fork[] forks = new Fork[n];

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

        for (int i=0; i<n; ++i) {
            philosophers[i] = new Philosopher(forks[i], forks[(i+1)%n], lock);
        }

        for (int i=0; i<n; ++i) {
            philosophers[i].start();
        }

        for (int i=0; i<n; ++i) {
            philosophers[i].join();
        }
    }
}
```

Tutaj problemem jest to, że gdy filozof chce podnieść widelce, blokuje wszystkich innch filozofów. W sytuacji, gdzie tylko jednen z widelców filozofa "podnoszącego" jest podniesiony, wszyscy inni filozofowie muszą czekać, pomimo że któryś z nich może mieć oba wolne widelce.

# 4 Rozwiązanie z lokajem

```
class Fork {
    private boolean available = true;

    public synchronized void pickUp() {
        while (!available) {
            try {
                wait();
```

```
            } catch (InterruptedException e) {
                System.exit(0);
            }
        }

        available = false;
    }

    public synchronized void putDown() {
        available = true;
        notifyAll();
    }

    public boolean isAvailable() {
        return this.available;
    }
}

class Philosopher extends Thread {
    private int counter = 0;
    private Fork rightFork;
    private Fork leftFork;
    private Waiter waiter;

    public Philosopher(Fork leftFork, Fork rightFork, Waiter waiter) {
        this.leftFork = leftFork;
        this.rightFork = rightFork;
        this.waiter = waiter;
    }

    public void run() {
        while (true) {

        thinkOrEat(500);

        waiter.giveForks(leftFork, rightFork);

        thinkOrEat(100);
        ++counter;
        if (counter % 100 == 0) {
            System.out.println("Philosopher: " + Thread.currentThread() +
                " have eaten " + counter + " times");
        }

        waiter.takeForks(leftFork, rightFork);
        }
```

```
    }

    public void thinkOrEat(long time) {
        try {
            Thread.sleep(time);
        } catch (InterruptedException e) {
            System.exit(0);
        }
    }
}

class Waiter {
    private int forksTaken = 0;

    public synchronized void giveForks(Fork leftFork, Fork rightFork) {
        while (!leftFork.isAvailable() || !rightFork.isAvailable())
            try {
                wait();
            } catch (InterruptedException e) {
                System.exit(0);
            }

        while (forksTaken == 4) {
            try {
                wait();
            } catch (InterruptedException e) {
                System.exit(0);
            }
        }

        forksTaken += 2;
        leftFork.pickUp();
        rightFork.pickUp();
    }

    public synchronized void takeForks(Fork leftFork, Fork rightFork) {
        forksTaken -= 2;
        leftFork.putDown();
        rightFork.putDown();
        notifyAll();
    }

}

public class Fil5mon {
    public static void main(String[] args) throws InterruptedException {
```

```
        int n = 5;

        Philosopher[] philosophers = new Philosopher[n];
        Fork[] forks = new Fork[n];

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

        Waiter waiter = new Waiter();

        for (int i=0; i<n; ++i) {
            philosophers[i] = new Philosopher(forks[i], forks[(i+1)%n], waiter);
        }

        for (int i=0; i<n; ++i) {
            philosophers[i].start();
        }

        for (int i=0; i<n; ++i) {
            philosophers[i].join();
        }
    }
}
```

W tym rozwiązaniu pojawia się nowa klasa - Lokaj, który "wydaje" filozofom widelce, ale tylko pod pewnymi warunkami, które gwarantują, że nie dojdzie do zakleszczenia (np. czeka, jeżeli jeden z widelców proszącego filozofa nie jest dostępny).

# 5 Porównanie

Pierwsze rozwiązanie bardzo często skutkuje szybkim zakleszczeniem, dlatego nie będzie bran w porównaniu wydajności. Poniżej znajduje się porównanie czasów działania pozostałych dwóch wariantów.

Oba warianty zostały przetestowane dla czasu myślenia wynoszącego 500 ms, oraz czasu jedzenia wynoszącego 100ms. Filozof kończył działanie, kiedy jego licznik wyniósł wartość 10, tak więc program kończył działanie, kiedy wszyscy filozofowie "zjedli 10 posiłków".

Dla wariantu z kelnerem średni wyznaczony czas wyniósł 6203 ms, natomiast dla wariantu drugiego 6321 ms. Wartości te są bardzo zbliżone, co może wynikać z tego, że oba rozwiązania sprowadzają się do tego, żeby zagwarantować to, że dany filozof dotanie dwa widelce.

# 6 Wnioski

Problem ucztujących filozofów pokazuje trudności w synchronizacji zasobów pomiędzy procesmi. W naiwnej implementacji problemu może dojść do zakleszczenia uniemożliwiającego dalsze działanie programu. Zakleszczeniu można zapobiec na różne spodoby, np. wprowadzając "lokaja", który dzięki informacji o obecnym stanie problemu ma możliwość stwierdzenia, czy wzięcie danego widelca przez pytającego filozofa mogłoby poskutkować zakleszczeniem.

# 7 Bibliografia


1. Dokumentacja języka Java - docs.oracle.com
2. Dijkstra, E. W. *Hierarchical ordering of sequential processes*