

# Laboratorium 5 - Problem pięciu filozofów

Piotr Karamon

12.11.2024r.

## Treści zadań

### Problem

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

### Zadania

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
5. Dodatkowe zadanie: Zaproponować autorskie rozwiązanie, inne niż w/w

### Zadanie a

Każdy filozof ma dostęp do dwóch widelców, które są reprezentowane przez obiekty klasy `Fork`. Widelce są zaimplementowane przy użyciu binarnego semafora, który jest zmienną

synchronizującą. Czas myślenia jest dłuższy niż czas jedzenia, jedzenie i myślenie jest zaimplementowane przy użyciu `Thread.sleep`. Gdy filozof chce zjeść, to próbuje podnieść oba widelce, najpierw lewy, a później prawy. Po zjedzeniu obiadu, filozof odkłada widelce, w kolejności odwrotnej do podnoszenia.

Kod klasy Fork:

```
class Fork {
    private final Semaphore _semaphore = new Semaphore(1);

    public void take() {
        try {
            _semaphore.acquire();
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }

    public void put() {
        _semaphore.release();
    }
}
```

Kod klasy Philosopher:

```
class Philosopher extends Thread {
    private final Fork _left;
    private final Fork _right;
    private final int _id;
    private int _counter = 0;

    public Philosopher(int id, Fork left, Fork right) {
        _id = id;
        _left = left;
        _right = right;
    }

    public void run() {
        while (true) {
            think();
            eat();
        }
    }

    private void think() {
        try {
            sleep(3);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    private void eat() {
```

```

        _counter++;
        _left.take();
        _right.take();
        try {
            sleep(1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.printf("Filozof: %d jadłem %d razy%n", _id, _counter);

        _right.put();
        _left.put();
    }
}

```

Kod uruchamiający:

```

public static void main(String[] args) {
    int N = 5;
    Fork[] forks = new Fork[N];
    Philosopher[] philosophers = new Philosopher[N];

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

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

    try {
        for (int i = 0; i < N; i++) {
            philosophers[i].join();
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

```

Wynik programu:

```

...
Filozof: 4 jadłem 919 razy
Filozof: 3 jadłem 915 razy
Filozof: 1 jadłem 920 razy
...
Filozof: 5 jadłem 8405 razy
Filozof: 4 jadłem 8406 razy
Filozof: 3 jadłem 8394 razy
Filozof: 2 jadłem 8413 razy
Filozof: 1 jadłem 8407 razy
Filozof: 5 jadłem 8406 razy
Filozof: 4 jadłem 8407 razy
Filozof: 3 jadłem 8395 razy
Filozof: 1 jadłem 8408 razy
Filozof: 5 jadłem 8407 razy
Filozof: 4 jadłem 8408 razy
Filozof: 2 jadłem 8414 razy

```

**Wnioski:** Co ciekawe program przez bardzo długi czas zdaje się działać poprawnie i nie zachodzi zakleszczenie. Filozofie jedzą w bardzo podobnym tempie, żaden z nich nie jest zagłodzony. Każdy filozof zdołał zjeść > 8000 razy, zanim nastąpiło całkowite zakleszczenie programu. Przykład pokazuje jak bardzo niepewnym sposobem testowania kodu współbieżnego jest sama obserwacja wyników programu i tego czy zachodzi zakleszczenie. Program potrzebował 1 minuty aby nastąpiło zakleszczenie. Łatwo wyobrazić sobie sytuację, w której ktoś uruchomiłby program na 10/20 sekund i uznałby go za poprawny.

## Zadanie b

W tej wersji filozofowie podnoszą oba widelce jednocześnie. Jeżeli jeden z widelców jest zajęty to filozof czeka(odkładając widelec, jeżeli udało mu się jakiś podnieść) i próbuje ponownie za pewien czas.

Dodajemy do klasy Fork metodę `tryTake`, która zwraca `true` jeżeli udało się podnieść widelec.

```

public boolean tryTake() {
    return _semaphore.tryAcquire();
}

```

Zmieniamy metodę `eat` w klasie `Philosopher`:

```

private void eat() {
    do {
        if (_left.tryTake()) {
            if (_right.tryTake()) {

```

```

        break;
    } else {
        _left.put();
    }
}
} while (true);

_counter++;
try {
    sleep(1);
} catch (InterruptedException e) {
    e.printStackTrace();
}
System.out.printf("Filozof: %d jadłem %d razy%n", _id, _counter);
_right.put();
_left.put();
}

```

W wyniku uruchomienia programu nie zostały wykryte zakleszczenia, nawet w przypadku wyrzucenia wywołań `sleep`. Rozwiązanie to nie powoduje zakleszczeń, ponieważ nie jest spełniony warunek *hold and wait*. Każdy filozof gdy nie może wziąć drugiego widelca to opuszcza pierwszy, pozwalając innym filozofom na podniesienie widelców. Takie rozwiązanie natomiast może prowadzić do zagłodzenia. Jeżeli jeden z filozofów będzie miał bardzo "łakomych" sąsiadów, którzy będą bardzo często jeść to nie będzie on w stanie uzyskać dwóch widelców potrzebnych do jedzenia.

## Zadanie c

W tej wersji filozofowie mają dostęp do lokaja. Lokaj jest reprezentowany przez obiekt klasy `Butler`. Lokaj ma dwie metody `acquire` i `release`. Lokaj da o to, aby w każdej chwili co najwyżej czterech filozofów próbowało podnieść widelce.

Lokaj w gruncie rzeczy jest semaforem licznikowym o początkowej wartości 4.

Kod klasy `Butler`:

```

class Butler {
    private final Semaphore _semaphore = new Semaphore(4);

    public void acquire() {
        try {
            _semaphore.acquire();
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }

    public void release() {
        _semaphore.release();
    }
}

```

```
}
```

W klasie `Philosopher` zmieniamy metodę `eat`, by uwzględniała lokaja:

```
private void eat() {
    _butler.acquire();
    _left.take();
    _right.take();

    _counter++;
    try {
        sleep(1);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.printf("Filozof: %d jadłem %d razy%n", _id, _counter);
    _right.put();
    _left.put();
    _butler.release();
}
```

W wyniku uruchomienia programu nie zostały wykryte zakleszczenia, nawet w przypadku wyrzucenia wywołań `sleep`. Rozwiązanie nie powoduje zakleszczeń, ponieważ lokaj gwarantuje, że w każdej chwili co najwyżej czterech filozofów próbuje podnieść widelce. W takim przypadku zawsze co najmniej jeden filozof będzie mógł podnieść oba widelce i zjeść. Rozwiązanie to również nie prowadzi do zagłodzenia.

## Pomiary

W celu wykonania pomiarów zmieniamy klasę `Philosopher`, by filozof jadł maksymalnie 100 razy. Będziemy mierzyć ile czasu filozof potrzebuje od momentu chęci jedzenia (zanim zacznie wołać lokaja/podnosić widelce, czyli sam początek metody `eat`) do momentu rozpoczęcia jedzenia. Wyniki pomiarów będą uśrednione dla 25 prób dla każdego z filozofów. Usuwamy również wywołania `println` z metody `eat`. Musimy uważać na testowanie wariantu pierwszego, ponieważ może dojść do zakleszczenia i program się nie wykona, zastosujemy więc ograniczenie czasowe na 1 minutę.

Kod klasy `Philosopher` musiał być zmodyfikowany dla każdego z wariantów. Ale schemat zmian wygląda tak:

```
private long waitingTime = 0; // nowe pole na czas czekania

public void run() {
    while(_counter < 100) {
        think();
        eat();
    }
}
```

```

private void eat() {
    _counter++;
    long start = System.currentTimeMillis();

    // interakcja z lokajem, zajęcie widelców

    long endTime = System.currentTimeMillis();
    waitingTime += endTime - startTime;

    // jedzenie + odkładanie widelców
}

public long getWaitingTime() {
    return waitingTime;
}

```

## Wariant a - z możliwym zakleszczeniem

Kod testujący:

```

public static void main(String[] args) {
    var times = new ArrayList<List<Long>>();
    IntStream.range(0, 25).forEach(i -> {
        if (testCase().isPresent()) {
            times.add(testCase().get());
        }
    });

    var avgTimes = new ArrayList<Long>();
    for (int i = 0; i < 5; i++) {
        long sum = 0;
        for (int j = 0; j < times.size(); j++) {
            sum += times.get(j).get(i);
        }
        avgTimes.add(sum / times.size());
    }

    IntStream.range(1, 6).forEach(i -> System.out.printf("%d, %d ms%n", i, avgTimes.get(i)
        ↪ - 1));
}

private static Optional<List<Long>> testCase() {
    int N = 5;
    Fork[] forks = new Fork[N];
    Philosopher[] philosophers = new Philosopher[N];
    for (int i = 0; i < N; i++) {
        forks[i] = new Fork();
    }
    for (int i = 0; i < N; i++) {
        philosophers[i] = new Philosopher(i + 1, forks[i], forks[(i + 1) % N]);
    }
}

```

```

var executor = Executors.newFixedThreadPool(N);
Arrays.stream(philosophers).forEach(executor::submit);
executor.shutdown();

boolean finishedOk = false;
try {
    finishedOk = executor.awaitTermination(1, TimeUnit.MINUTES);
} catch (InterruptedException e) {
    e.printStackTrace();
}

if (!finishedOk) {
    executor.shutdownNow();
    return Optional.empty();
} else {
    return Optional.of(
        Arrays
            .stream(philosophers)
            .map(Philosopher::getWaitingTime)
            .toList());
}
}

```

Wynik:

```

1, 97 ms
2, 100 ms
3, 101 ms
4, 101 ms
5, 98 ms

```

## Wariant b - z możliwym zagłodzeniem

Kod testujący jest analogiczny do poprzedniego, jedynie jest trochę prostszy z racji gwarancji zakończenia.

Wynik:

```

1, 33 ms
2, 33 ms
3, 32 ms
4, 31 ms
5, 34 ms

```



## Wariant c - z lokajem

Wynik:

1, 23 ms  
2, 17 ms  
3, 19 ms  
4, 22 ms  
5, 22 ms

## Wykres



## Wnioski

We wszystkich rozwiązaniach średni czas oczekiwania był podobny dla każdego z filozofów. Wynika to z faktu, że wartości przekazywane do funkcji `sleep` są stałe, co oznacza że każdy z filozofów pracuje w podobnym tempie i podobnie często próbuje podnieść widelce. Rozwiązanie trywialne jest nie tylko najmniej wydajne, ale również oczywiście błędne. Najszybszym rozwiązaniem okazało się rozwiązanie z lokajem, co jest zgodne z oczekiwaniami.

## Bibliografia

- Problem uczujących filozofów
- [https://wazniak.mimuw.edu.pl/index.php?title=Programowanie\\_wsp%C3%B3%C5%82bie%C5%BCne\\_i\\_rozproszone/PWR\\_Wyk%C5%82ad\\_9](https://wazniak.mimuw.edu.pl/index.php?title=Programowanie_wsp%C3%B3%C5%82bie%C5%BCne_i_rozproszone/PWR_Wyk%C5%82ad_9)