# Laboratorium 1 - Współbieżność w Javie

Piotr Karamon

14.10.2024r.

## Treści zadań

### Zadanie 1

Napisać program (szkielet), który uruchamia 2 wątki, z których jeden zwiększa wartość zmiennej całkowitej o 1, drugi wątek zmniejsza wartość o 1. Zakładając że na początku wartość zmiennej Counter była 0, chcielibyśmy wiedzieć jaka będzie wartość tej zmiennej po wykonaniu 10000 operacji zwiększania i zmniejszania przez obydwa wątki.

### Zadanie 2

Na podstawie 100 wykonań programu z p.1, stworzyć histogram końcowych wartości zmiennej Counter.

### Zadanie 3

Spróbować wprowadzić mechanizm do programu z p.1, który zagwarantowałby przewidywalną końcową wartość zmiennej Counter. Nie używać żadnych systemowych mechanizmów, tylko swój autorski.

### Zadanie dodatkowe

W systemie działa N wątków, które dzielą obiekt licznika (początkowy stan licznika = 0).

Każdy wątek wykonuje w pętli 5 razy inkrementację licznika. Zakładamy, że inkrementacja składa się z sekwencji trzech instrukcji: `read`, `inc`, `write` (odczyt z pamięci, zwiększenie o 1, zapis do pamięci). Wątki nie są synchronizowane.

1. Jaka jest teoretycznie najmniejsza wartość licznika po zakończeniu działania wszystkich wątków i jaka kolejność instrukcji (przeplot) do niej prowadzi?

2. Spróbować znaleźć dowód, że będzie to zawsze najmniejsza wartość.

## Zadanie 1

W celu wykonania eksperymentu tworzymy dwie klasy, które dziedziczą po klasie `Thread`. Nadpisujemy w nich metodę `run` gdzie umieszczamy logikę jaką chcemy by nasze wątki wykonywały. Wątki uruchamiamy metodą `start()`. By sprawdzić końcową wartość zmiennej `Counter` musimy zaczekać aż wątki skończą pracę, używamy do tego metody `join()`.

```
// Race.java
// Wyscig

class Counter {
    private int _val;

    public Counter(int n) {
        _val = n;
    }

    public void inc() {
        _val++;
    }

    public void dec() {
        _val--;
    }

    public int value() {
        return _val;
    }
}

// Watek, ktory inkrementuje licznik 100.000 razy
class IThread extends Thread {
    private final Counter counter;

    public IThread(Counter counter) {
        this.counter = counter;
    }

    @Override
```

```
    public void run() {
        for (int i = 0; i < 100_000; i++) {
            counter.inc();
        }
    }
}

// Watek, ktory dekrementuje licznik 100.000 razy
class DThread extends Thread {
    private final Counter counter;

    public DThread(Counter counter) {
        this.counter = counter;
    }

    @Override
    public void run() {
        for (int i = 0; i < 100_000; i++) {
            counter.dec();
        }
    }
}

public class Race {

    public static void main(String[] args) {
        Counter cnt = new Counter(0);

        IThread incThread = new IThread(cnt);
        DThread decThread = new DThread(cnt);

        incThread.start();
        decThread.start();

        try {
            incThread.join();
            decThread.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("stan=" + cnt.value());
    }
}
```

W wyniku uruchomienia otrzymujemy następujące wyjście:

```
stan=1955
```

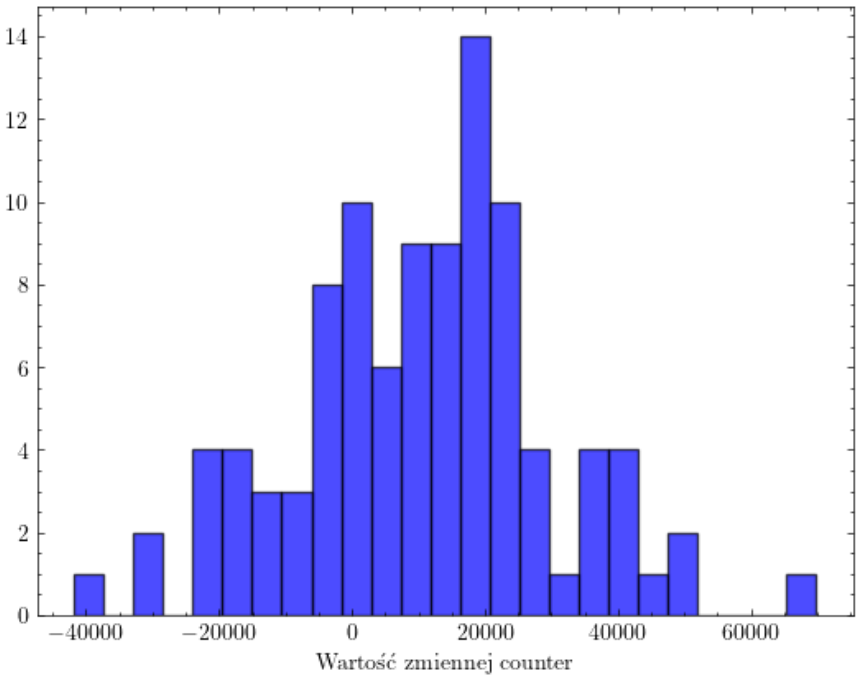
**Wnioski**: Jest to wynik znacznie odbiegający od spodziewanego zera. Mamy tutaj do czynienia z wyścigiem, czyli sytuacją przy której więcej niż jeden wątek korzysta jednocześnie z zasobu współdzielonego, przy czym co najmniej jeden próbuje go zmienić. Wyścig sprawia, że nasz program staje się niedeterministyczny. Wątki wykonują operacje zwiększania i zmniejszania na zmiennej `Counter` bez synchronizacji. Może to prowadzić do sytuacji, w której jeden wątek odczytuje wartość Counter, podczas gdy drugi wątek ją zmienia, co prowadzi do utraty niektórych operacji.

## Zadanie 2

Program z zadania 1 uruchamiamy 100 razy w celu stworzenia histogramu. Korzystamy z powłoki `bash` oraz prostej komendy w celu zebrania danych.

```
for i in {1..100}; do ./gradlew run |  grep -E -o 'stan=.*' | sed
↪  's/stan=//' >> output.txt; done
```

Z otrzymanych danych w pliku `output.txt` tworzymy histogram.




**Rysunek 1:** Histogram powstały ze 100 uruchomień programu z zadania 1.

**Wnioski**: Widzimy bardzo duży rozrzut wartości. Co ciekawe rozkład zdaje się być niesymetryczny względem zera. Histogram pokazuje fakt, iż obecnie bez żadnej synchronizacji nasz program działa w bardzo losowy i nieprzewidywalny sposób. Jednakże zgodnie z naszą intuicją najwięcej wyników znajduje się blisko zera.

## Zadanie 3

Celem zadania jest wprowadzenie autorskiego mechanizmu do programu z zadania 1, który zagwarantuje nam deterministyczny wynik.

To co możemy zrobić to wymuszenie kolejności typu: inc, dec, inc, dec, itd. Do tego celu potrzebujemy wspólnej zmiennej:

```
public static volatile int turn = 0;
```

Wartość 0 oznacza, że wątek inkrementujący ma wykonać jedno `inc()`. Wartość 1 oznacza, że wątek dekrementujący ma wykonać jedno `dec()`. Korzystamy tutaj ze słowa kluczowego `volatile`. Dzięki niemu zmiany tej

zmiennej są natychmiast widoczne dla innych wątków. To słowo zapobiega również optymalizacji polegającej na pobraniu tej zmiennej z pamięci cache.

Potrzebna jest zmiana metod `run()` w naszych wątkach. Będziemy po prostu czekać w pętli, aż nie najdzie na nas kolej.

```
class IThread extends Thread {
    private final Counter counter;

    public IThread(Counter counter) {
        this.counter = counter;
    }

    @Override
    public void run() {
        for (int i = 0; i < 100_000; i++) {
            while(true) {
                if (Race.turn == 0) {
                    counter.inc();
                    Race.turn = 1;
                    break;
                }
            }

        }
    }
}

class DThread extends Thread {
    private final Counter counter;

    public DThread(Counter counter) {
        this.counter = counter;
    }

    @Override
    public void run() {
        for (int i = 0; i < 100_000; i++) {
            while(true) {
                if (Race.turn == 1) {
                    counter.dec();
                    Race.turn = 0;
                    break;
                }
            }
        }
    }
}
```

Testujemy nasze rozwiązanie uruchamiając je 200 razy.

```
public static void main(String[] args) {
    var allZeros = IntStream.range(0, 200).map(i ->
    ↪  runSample()).allMatch(i -> i == 0);
    System.out.println("all zeros = "+allZeros);
}

private static int runSample() {
    Counter cnt = new Counter(0);

    Race.turn = 0;

    IThread incThread = new IThread(cnt);
    DThread decThread = new DThread(cnt);

    incThread.start();
    decThread.start();

    try {
            incThread.join();
            decThread.join();
    } catch (InterruptedException e) {
            e.printStackTrace();
    }

    return cnt.value();
}
```

Otrzymane wyjście:

```
all zeros = true
```

**Wnioski**: Nasze rozwiązanie zdaje się działać, jednakże jest ono wysoce nieefektywne głównie dlatego, że wątki używają pętli `while (true)` do oczekiwania na zmianę stanu zmiennej `turn`. To prowadzi do "busy waiting", gdzie wątki nie wykonują żadnej użytecznej pracy, a jedynie ciągle sprawdzają stan zmiennej. To jest bardzo nieefektywne z perspektywy zasobów procesora. Dlatego realistycznie powinniśmy skorzystać z gotowych mechanizmów takich jak:

- bloki/funkcje `synchronized`
- `ReentranLock`
- `Semaphore`
- zmienne atomowe w tym przypadku `AtomicInteger`

## Zadanie dodatkowe

Idea rozwiązania polega na wczesnym wykonaniu instrukcji `read` oraz `inc` przez jeden wątek, następnie na pozwoleniu na pracę innych wątków, których praca zostanie nadpisana po wywołaniu `write` przez ten wczesny wątek.

Dla $N = 1$ problem staje się całkowicie sekwencyjny, zatem otrzymamy wartość równą 5. Dla $N > 1$ najmniejsza możliwa wartość licznika to 2. Odpowiadający przeplot:

| wątek | instrukcja |
|---|---|
| $w_1$ | read |
| $w_1$ | inc |
| $w_3..w_N$ | wykonują pełne 5 iteracji |
| $w_2$ | wykonuje pełne 4 iteracji |
| $w_1$ | write(teraz `counter=1`) |
| $w_2$ | read(wczytanie `counter=1`) |
| $w_1$ | wykonuje pozostałe 4 iteracje w całości |
| $w_2$ | inc |
| $w_2$ | write(wpisanie `counter=2`) |

Najmniejsza możliwa wartość licznika nie może być równa 0, ponieważ niezależnie od przeplotu końcową instrukcją jest `write`, odpowiadający mu `read` (ten sam wątek, ta sama iteracja) wczytał liczbę która jest $\geq 0$, przez to dzięki `inc` wpiszemy wartość $>= 1$.

Teraz zastanówmy się, czy możliwe jest by licznik miał na końcu wartość 1. Zrobimy dowód nie wprost.

Jeżeli na końcu dostaliśmy wartość 1, to ostatnia instrukcja `write`, wpisała wartość 1(niech wykona ją wątek $w_a$). Odpowiadający tej instrukcji `read` (ten sam wątek, ta sama iteracja) musiał wczytać 0. To oznacza, że żaden wątek nie mógł wcześniej niż ta instrukcja `read` wykonać instrukcji `write` (bo wczytana wartość byłaby $> 0$). Ale to oznacza, że również wątek $w_a$ nie mógł wykonać instrukcji `write`, co jest sprzecznością, ponieważ w momencie wykonania ostatniej swojej instrukcji `read` jest w 5 iteracji, a co za tym idzie wykonał on 4 instrukcje `write`.

**Wnioski**: Możliwe przeploty potrafią prowadzić do bardzo zaskakujących wyników. Opisany powyżej przeplot jest kompletnie nieintuicyjny oraz bardzo mało prawdopodobny, fakt jego istnienia pokazuje trudność programowania współbieżnego oraz analizy algorytmów współbieżnych.

## Bibliografia


- Bill Venners: *Inside the Java Virtual Machine Chapter 20*
- Atomic Access
- Dokumentacja klasy thread