Laboratorium I

Współbieżność w Javie

Dominik Marek

7 października 2024

AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA
W KRAKOWIE

1. Zadania

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.

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

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

4. Napisać sprawozdanie z realizacji pp. 1-3, z argumentacją i interpretacją wyników.

2. Rozwiązania

1.

Rozwiązywanie zadań rozpocząłem od zapoznania się z metodami implementacji i zarządzania wątkami w Javie. Następnie przystąpiłem do uzupełniania kodu. W klasach IThread oraz DThread odpowiedzialnych odpowiednio za inkrementację i dekrementację licznika dodałem atrybut counter, który będzie ustawiany w konstruktorze wyżej wymienionych klas. Kolejno zgodnie z dokumentacją Javy dla metod rozszerzających klasę Thread stworzyłem metodę run() , w której odpowiednio dla danej klasy licznik jest zwiększany bądź zmniejszany 10 000 razy. Następnie w klasie Race tworzę instancję klasy inkrementującej(iThread) i dekrementującej(dThread) licznik i wywołuję na nich metodę start(), która uruchamia wątki. Finalnie na obiektach iThread i dThread wywołuję metodę join() aby główny wątek zaczekał na ich zakończenie i wypisuję na standardowe wyjście aktualny stan licznika.

```
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 10.000 razy
class IThread extends Thread {
    private Counter counter;

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

    }

    public void run() {
        for (int i = 0; i < 10000; i++) {
            counter.inc();
        }

    }
```

```
}


// Watek, ktory dekrementuje licznik 10.000 razy
class DThread extends Thread {

    private Counter counter;

    public DThread (Counter counter) {

        this.counter = counter;
    }

    public void run(){
        for (int i  = 0 ; i < 10000 ; i ++){
            counter.dec();
        }

    }

}

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

        IThread iThread = new IThread(cnt);
        DThread dThread = new DThread(cnt);

        iThread.start();
        dThread.start();


        try {
            iThread.join();
            dThread.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

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

Dla pięciu kolejnych uruchomień powyższego kodu otrzymałem następujące wartości licznika:

374, -992, 2015, -125 oraz 524.

Wnioski:

Analizując powyższe wyniki można zauważyć, iż po mimo jednakowej liczby inkrementacji i dekrementacji licznika finalna wartość różni się od zakładanej wartości 0. Dzieje się tak , gdyż powyższy kod nie gwarantuje poprawnej synchronizacji między wątkami, ponieważ operacje na liczniku [inc() i dec()] nie są atomowe. W praktyce mamy tu do czynienia z tzw. Warunkiem wyścigu (race condition), co może prowadzić do błędnych wyników, gdyż oba wątki mogą

próbować jednocześnie uzyskać dostęp i modyfikować współdzielony zasób, w naszym przypadku jest to wartość licznika.

W Javie, operacje takie jak inkrementacja (_val++) i dekrementacja (_val--) są nieatomowe, co oznacza, że składają się z kilku kroków:

I. Odczyt wartości zmiennej _val.

II. Zwiększenie (lub zmniejszenie) wartości.

III. Zapis zaktualizowanej wartości z powrotem do zmiennej _val.

Jeśli dwa wątki wykonają te operacje jednocześnie, mogą przeplatać swoje kroki, co prowadzi do sytuacji, w której zmienna _val może zostać nadpisana, a modyfikacje jednego wątku mogą zostać „utracone". Może to powodować, iż po mimo równej liczby inkrementacji i dekrementacji końcowa wartość licznika nie jest równa 0.

W celu uniknięcia zjawiska race condition i zapewnienia, że końcowa wartość licznika jest poprawna (tj. wynosi 0), należy zsynchronizować operacje na współdzielonym zasobie. Można to osiągnąć na przykład , używając słowa kluczowego synchronized w metodach inc() i dec(), aby zapewnić, że tylko jeden wątek naraz będzie miał dostęp do modyfikowania wartości _val.

2.

W celu uzyskania 100 wywołań programu z zadania 1 wproawdziłem następujące modyfikację w kodzie:

Do klasy Race dodałem argument raceNumber, ustawiany w konstruktorze klasy, który odpowiada za ilość wyołań logiki z pierwszego zadania.Dodatkowo stworzyłem hash mapę counterValueMap, którą używam do zliczenia ilości wstąpień poszczególnych wyników finalnego stanu licznika.

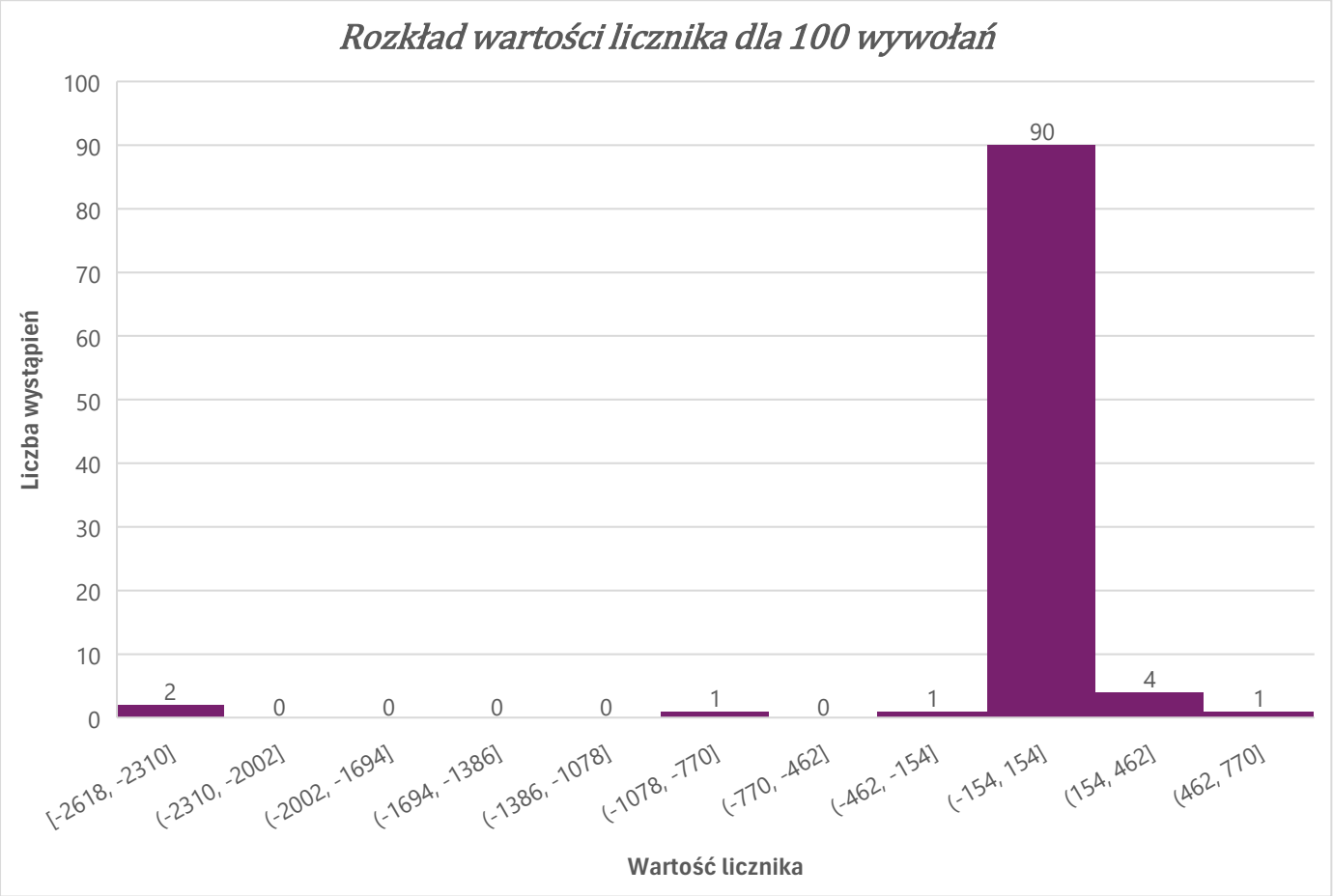
```
public class Race {

    private final int numberOfRun;

    Map<Integer, Integer> counterValueMap =  new HashMap<>();

    public Race(int raceNumber) {

        this.numberOfRun = raceNumber;

    }

    public void beginRace(){
        for(int i = 0; i < numberOfRun; i ++){
            Counter cnt = new Counter(0);

            IThread iThread = new IThread(cnt);
            DThread dThread = new DThread(cnt);

            iThread.start();
            dThread.start();
```

```
            try {
                iThread.join();
                dThread.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            counterValueMap.put(cnt.value(),
counterValueMap.getOrDefault(cnt.value(), 0) + 1);


        }
        for (Map.Entry<Integer, Integer> entry :
counterValueMap.entrySet()) {
            System.out.println("Wartość " + entry.getKey() + ": " +
entry.getValue() + " razy");
        }


    }

}

public class Main {
    public static void main(String[] args) {
        Race race = new Race(100);
        race.beginRace();
    }
}
```

Po wykonaniu powyższego kodu otrzymujemy następujące wyniki przedstawione w poniższej tabeli.

Wartość licznika	***Liczba wystąpień***
0	88
-1537	1
257	1
227	1
260	1
-82	1
-2580	1
726	1
-89	1
185	1
-2618	1
-158	1
-831	1

Każdy z otrzymanych wyników został przypisany do jednego z 11 przedziałów każdy o szerokości wynoszącej 308, liczność każdego z przedziałów została przedstawiona na poniższym histogramie.



Histogram wyników pokazuje, że w większości przypadków (90 na 100) końcowa wartość licznika mieści się w przedziale od -154 do 154, z czego 88 wyników ma wartość 0. Potwierdza to , że w większości sytuacji na liczniku poprawnie wykonuje się zbliżona ilość inkrementacji i dekrementacji, co oznacza, że race condition nie zawsze prowadzi do widocznych błędów. Jednak wartość zerowa nie jest gwarantowana, co pokazują inne, rozproszone wyniki. Pojedyncze, skrajne wartości, takie jak -2580 czy 726, są wynikiem race condition, gdzie operacje inkrementacji i dekrementacji przeplatają się w niekontrolowany sposób. Brak synchronizacji powoduje, że program działa niedeterministycznie, a wynik licznika odbiega od oczekiwanej wartości .Synchronizacja zapewniłaby, że licznik zawsze kończyłby z wartością 0, eliminując te skrajne przypadki.

3.

W celu osiągnięcia przewidywalnej końcowej wartości zmiennej counter tworzę klasę PredictableCounter, dziedziczącą po klasie Counter. W nowo utworzonej klasie dodaje zmienną isDecrementing(początkowo ustawioną na false) która kontroluje, czy aktualnie wykonywana jest operacja dekrementacji . Powyższa zmienna jest typu AtomicBoolean , co umożliwia wykonywanie atomowych operacji na wartości logicznej (true/false). Dzięki temu modyfikacja zmiennej jest nieprzerywalna, tzn. nie może zostać zakłócona przez inny wątek. Pozwala to na bezpieczne i szybkie zarządzanie współdzielonym stanem między wątkami.

W metodzie inc() dopóki trawa dekrementacji licznika(zmienna isDecrementing ma wartość true) wątek czeka na jej zakończenie , a gdy się zakończy zwiększa wartość licznika i ustawia wartość zmiennej isDecrementing na true aby zasygnalizować , iż zakończył proces zwiększania licznika. Analogicznie postępujemy w metodzie dec(), gdzie jeśli wartość licznika jest inkrementowana to czekamy na zakończenie tej operacji, aby następnie dekrementować licznik i zasygnalizować o wykonaniu tej operacji zmieniając wartość zmiennej isDecrementing na false.

```
import java.util.concurrent.atomic.AtomicBoolean;

public class PredictableCounter extends Counter {

    private int _val;
    final AtomicBoolean isDecrementing = new AtomicBoolean(false);
    public PredictableCounter(int n) {
        super(n);
    }
    @Override
    public void inc() {
        while (isDecrementing.get()) {}
        _val++;
        isDecrementing.set(true);

    }

    @Override
    public void dec() {
        while (!isDecrementing.get()) {}
        _val--;
        isDecrementing.set(false);

    }
}
```

Po wykonaniu powyższego kodu dla 100 kolejnych przebiegów otrzymuje zawsze poprawną końcową wartość licznika równą zero.

Wnioski:

Dzięki zastawaniu operacji atomowych AtomicBoolean, wątki inkrementujący i dekrementujący nie mogą modyfikować licznika jednocześnie. Każdy z nich czeka, aż drugi zakończy swoją operację, zanim sam przejdzie do modyfikacji zmiennej _val. Dzięki czemu możliwe jest wyeliminowanie zjawiska race condition i otrzymanie deterministycznego programu , który daje przewidywalne wartości licznika.

3. Bibliografia

- https://www.dz5.pl/ti/java/java_skladnia.pdf ~Jacek Rumiński, Język Java – rozdział o wątkach
- https://www.artima.com/insidejvm/ed2/threadsynch.html ~Bill Venners, Inside the Java Virtual Machine -Charter 20 Thread Synchronization
- https://jenkov.com/tutorials/java-util-concurrent/atomicboolean.html
- https://docs.oracle.com/en/java/javase/19/index.html