Laboratorium 1

Andrzej Ratajczak

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.

```
public class Main {

    static int counter = 0;

    public static void main(String[] args) {

        Thread t1 = new Thread(new Runnable() {
            public void run() {
                for(int i = 0; i < 10000; i++) {
                    counter++;
                }
            }
        });

        Thread t2 = new Thread(new Runnable() {
            public void run() {
                for(int i = 0; i < 10000; i++) {
                    counter--;
                }
            }
        });
        t1.start();
        t2.start();
        try{
            t1.join();
            t2.join();
        } catch(InterruptedException e) {
            System.out.println("Ooops, something went wrong " + e.getMessage());
        }
        System.out.println(counter);
    }
}
```

Kod źródłowy 1. Implementacja zadanego problemu w języku Java.

Powyższy kod przedstawia implementację zadanego problemu oraz wypisuje na standardowe wyjście wartość zmiennej counter po zakończeniu pracy wątków.

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

```
public class Main {

    static int counter = 0;

    public static void main(String[] args) {
        Map<Integer, Integer> map = new HashMap<>();

        for(int j = 0; j < 100; j++) {
            Thread t1 = new Thread(new Runnable() {
                public void run() {
                    for (int i = 0; i < 10000; i++) {
                        counter++;
                    }
                }
            });

            Thread t2 = new Thread(new Runnable() {
                public void run() {
                    for (int i = 0; i < 10000; i++) {
                        counter--;
                    }
                }
            });
            t1.start();
            t2.start();
            try {
                t1.join();
                t2.join();
            } catch (InterruptedException e) {
                System.out.println("Ooops, something went wrong " +
                                   e.getMessage());
            }
            if(map.get(counter) == null) {
                map.put(counter, 1);
            } else {

                map.put(counter, map.get(counter) + 1);
            }

            counter = 0;
        }

        map.entrySet().forEach(entry->{
            System.out.println(entry.getKey() + " " + entry.getValue());
        });
    }
}
```

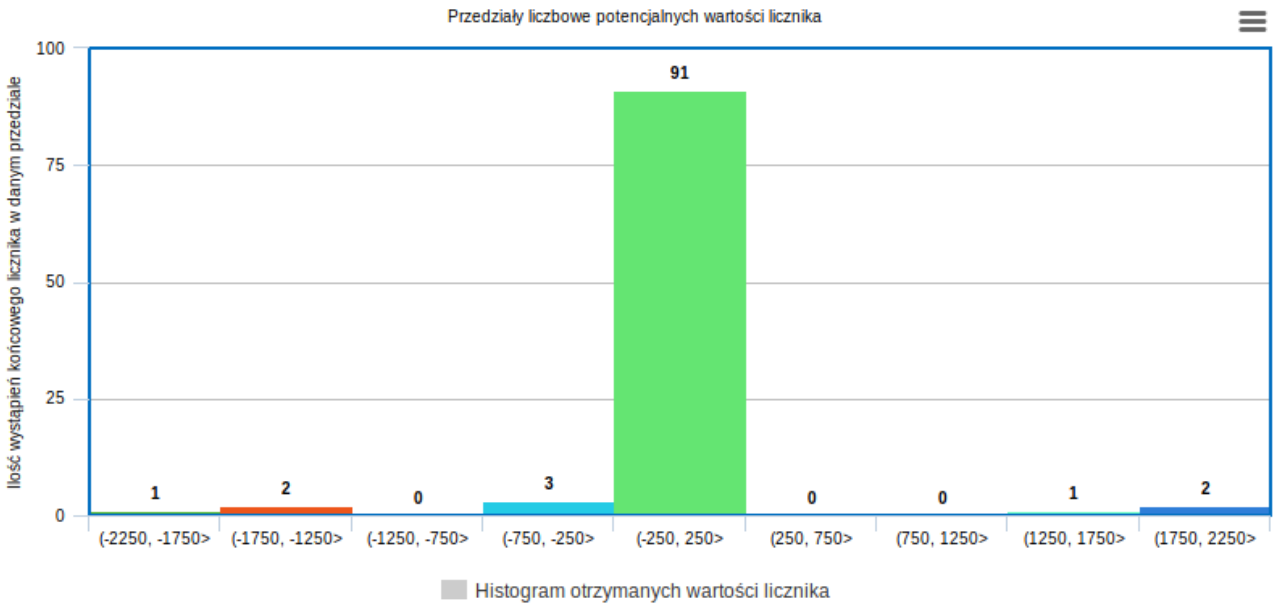
Kod źródłowy 2. Ulepszenie kodu źródłowego 1 o efektywne testowanie wątków.

Po przygotowaniu powyższego kodu uruchomiłem go testowo. Poniższa tabelka przedstawia przykładowy wynik na konsoli.

Otrzymana wrtość zmiennej *counter*	Ilość powtórzeń tej zmiennej
0	91
2099	1
-629	1
-1445	1
1686	1
-2060	1
-685	1
-1614	1
2077	1
-718	1

Tabela 1. Zebrane wyniki pojedynczego uruchomienia skompilowanego kodu źródłowego 2.

Jak widać nie wszystkie wyniki są równe zero tak jak byśmy się tego spodziewali. Powodem jest race condition. Oba wątki współbieżnie operują na jednej zmiennej, przez co jest możliwość zafałszowania wyniku. Dzieje się to dlatego, że oba wątki pobierają z rejestru wartość zmiennej counter, która w danym momencie przyjmuje wartość a. Następnie każdy z nich operuje na tej wartości (kolejno jeden wątek zmieni jej stan na a-1, drugi na a+1) oraz przechodzi do etapu zapisania obliczonej wartości do rejestru. W tym momencie jeden z nich (tutaj również nie mamy pewności w jakiej kolejności to się stanie) wpisze swoją wartość do rejestru (przykładowo a → a-1, następnie drugi również zapisze swoją wartość (a-1 → a+1). Mimo, że założenie pierwotne jest takie, że wykonanie operacji dodawania i odejmowania powinny się znieść i program po wykonaniu, powinien znaleźć się w takim samym stanie jak na początku, mamy zgoła inny wynik. Oczywiście nie kontrolujemy w żaden sposób wątków, więc wcześniej opisana sytuacja może mieć wiele innych kontynuacji: wątek pierwszy po wpisaniu stanu a-1 natychmiast pobiera tą wartość z rejestru i zaczyna na niej operować lub drugi wątek zdąży wpisać swoją wartość i pobierze a+1. Jedyny sposób by kontorolować operacje na zmiennej współdzielonej to wprowadzić mechanizm synchronizacji.



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

```
class Semaphore {

    private int mutex = 1;

    public void acquire() {
        try {
            Thread.sleep(0,10);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        while(mutex <= 0){
            try {
                Thread.sleep(0,10);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        mutex--;



    }

    public void release() {
        try {
            Thread.sleep(0,10);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        mutex++;
        try {
            Thread.sleep(0,10);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Kod źródłowy 3. Impelementacja semafora wirującego.

Aby zapewnić sobie bezpieczny dostęp do zmiennej współdzielonej stworzyłem własny semafor wirujący. Jego zasada jest bardzo prosta. Zmienna mutex pilnuje, aby do sekcji krytycznej wszedł tylko jeden wątek. Każdy kolejny wpadnie do nieskończonej pętli, która trwa tak długo, aż nie zmieni się globalny stan zmiennej mutex na dodatni.

Teraz wystaryczy otoczyć dostęp do zmienej współdzielonej metodami sekcji krytycznej.

```
Semaphore s = new Semaphore();
.
.
.
s.acquire();
counter++;
s.release();
```

Kod źrodłowy 4. Użycie semafora wirującego.

Użycie takiego semafora znacznie spowalnia działanie naszego programu z powodu usypiania wątków. Jednak jest to konieczne, aby zwiększyć szanse praktycznie do 100%, że dwa wątki nie zmienią stanu zmiennej mutex, ponieważ mogłoby to prowadzić do zakleszczeń lub powodować równoczesny dostęp do sekcji krytycznej.