Teoria Współbieżności - Sprawozdanie 1

Maciej Brzeżawski

Październik 2024

1 Treść 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 Implementacja wątków zwiększających i zmniejszających zmienną `counter`

W poniższym kodzie w języku Java został zaimplementowany program, który tworzy dwa wątki. Jeden z nich zwiększa wartość zmiennej `counter` o 1, a drugi zmniejsza `counter` o 1. Każdy z wątków wykonuje tę operację 10000 razy.

```
public class Main {
    private static int counter = 0;

    public static void main(String[] args) throws InterruptedException {
        Thread incrementThread = new Thread(() -> {
            for (int i = 0; i < 10000; i++) {
                counter++;
            }
        });

        Thread decrementThread = new Thread(() -> {
            for (int i = 0; i < 10000; i++) {
                counter--;
            }
        });

        incrementThread.start();
        decrementThread.start();

        incrementThread.join();
        decrementThread.join();

        System.out.println("Final value of counter: " + counter);
    }
}
```

3 Histogram końcowych wartości zmiennej `counter`

Na podstawie 100 wykonań programu z podpunktu pierwszego, stworzony został histogram przedstawiający końcowe wartości zmiennej `counter`. W tym celu w każdej iteracji program inkrementuje i dekrementuje zmienną `counter` po 10000 razy, a wynik końcowy jest zapisywany.

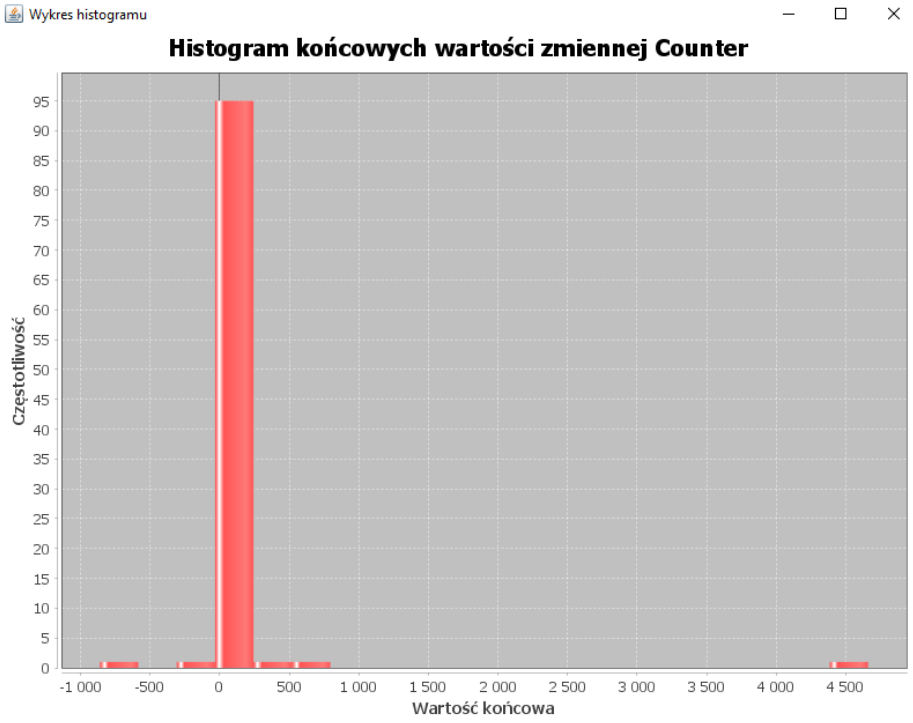
```
public static void main(String[] args) throws InterruptedException {
    int[] results = new int[100];

    for (int i = 0; i < 100; i++) {
        results[i] = simulateOperations();
    }

    HistogramPlotter.plotHistogram(results);
}

public static int simulateOperations() throws InterruptedException {
    final int[] counter = {0};

    Thread incrementThread = new Thread(() -> {
        for (int i = 0; i < 10000; i++) {
            counter[0]++;
        }
    });

    Thread decrementThread = new Thread(() -> {
        for (int i = 0; i < 10000; i++) {
            counter[0]--;
        }
    });

    incrementThread.start();
    decrementThread.start();

    incrementThread.join();
    decrementThread.join();

    System.out.println("Final value of counter: " + counter[0]);
    return counter[0];
}
```

Do wygenerowania histogramu została użyta biblioteka JFreeChart, która pozwala na tworzenie wykresów w Javie. Poniżej przedstawiony jest kod odpowiedzialny za wygenerowanie i wyświetlenie histogramu.

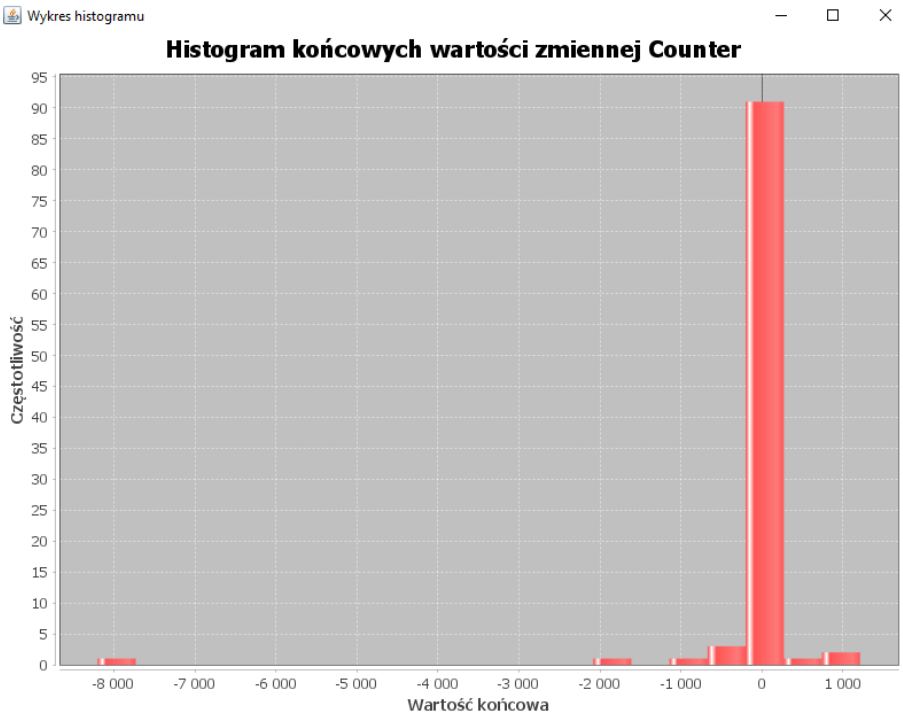
```
public class HistogramPlotter {
    public static void plotHistogram(int[] results) {
        double[] values = new double[results.length];
        for (int i = 0; i < results.length; i++) {
            values[i] = results[i];
        }

        HistogramDataset dataset = new HistogramDataset();
        dataset.addSeries("Counter Values", values, 20);

        JFreeChart histogram = ChartFactory.createHistogram(
            "Histogram końcowych wartości zmiennej Counter",
            "Wartość końcowa",
            "Częstotliwość",
            dataset,
            PlotOrientation.VERTICAL,
            false,
            false,
            false
        );

        ChartPanel chartPanel = new ChartPanel(histogram);
        chartPanel.setPreferredSize(new Dimension(800, 600));
        JFrame frame = new JFrame("Wykres histogramu");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add(chartPanel);
        frame.pack();
        frame.setVisible(true);
    }
}
```

3.1 Wyniki i analiza histogramu końcowych wartości zmiennej `counter`

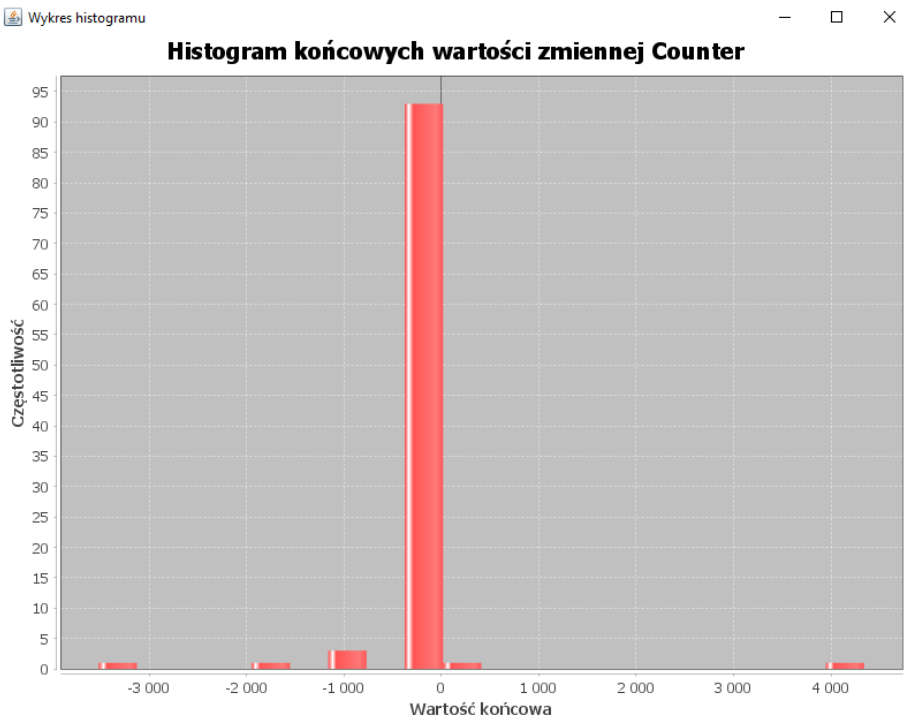
W wyniku braku synchronizacji między wątkami, w niektórych przypadkach wynik końcowy odbiegał od zera. Przyczyną tego są warunki wyścigu (ang. *race conditions*), gdzie dwa wątki jednocześnie próbują modyfikować wartość zmiennej `counter`, co prowadzi do nieprzewidywalnych wyników.



Rysunek 1: Histogram końcowych wartości zmiennej `counter` po 100 iteracjach



Rysunek 2: Histogram końcowych wartości zmiennej `counter` po 100 iteracjach



Rysunek 3: Histogram końcowych wartości zmiennej `counter` po 100 iteracjach

4 Autorski mechanizm synchronizacji

W poniższym kodzie zaimplementowano mechanizm naprzemiennego zwiększania i zmniejszania zmiennej `counter` przez dwa wątki, który próbuje działać bez użycia systemowych mechanizmów synchronizacji takich jak `wait()` czy `synchronized()`. Każdy wątek oczekuje na swoją kolej na podstawie flagi `isIncrementTurn`, która kontroluje, czy to wątek inkrementujący, czy dekrementujący powinien działać. Program zawiesza się lub nie działa zgodnie z oczekiwaniami, ponieważ wątki nie są synchronizowane za pomocą żadnych systemowych mechanizmów. Wykorzystanie jedynie flagi `isIncrementTurn` i pętli oczekującej (ang. *busy-waiting*) nie jest wystarczające do zagwarantowania poprawnego działania wątków, szczególnie w środowisku wielowątkowym.

```
public class Main {
    private static int counter = 0;
    private static boolean isIncrementTurn = true;

    public static void main(String[] args) throws InterruptedException {
        int[] results = new int[100];

        for (int i = 0; i < 100; i++) {
            results[i] = simulateOperations();
        }

        HistogramPlotter.plotHistogram(results);
    }

    public static int simulateOperations() throws InterruptedException {
        counter = 0;

        Thread incrementThread = new Thread(() -> {
            for (int i = 0; i < 10000; i++) {
                while (!isIncrementTurn) {

                }
                counter++;
                isIncrementTurn = false;
            }
        });

        Thread decrementThread = new Thread(() -> {
            for (int i = 0; i < 10000; i++) {
                while (isIncrementTurn) {

                }
                counter--;
```

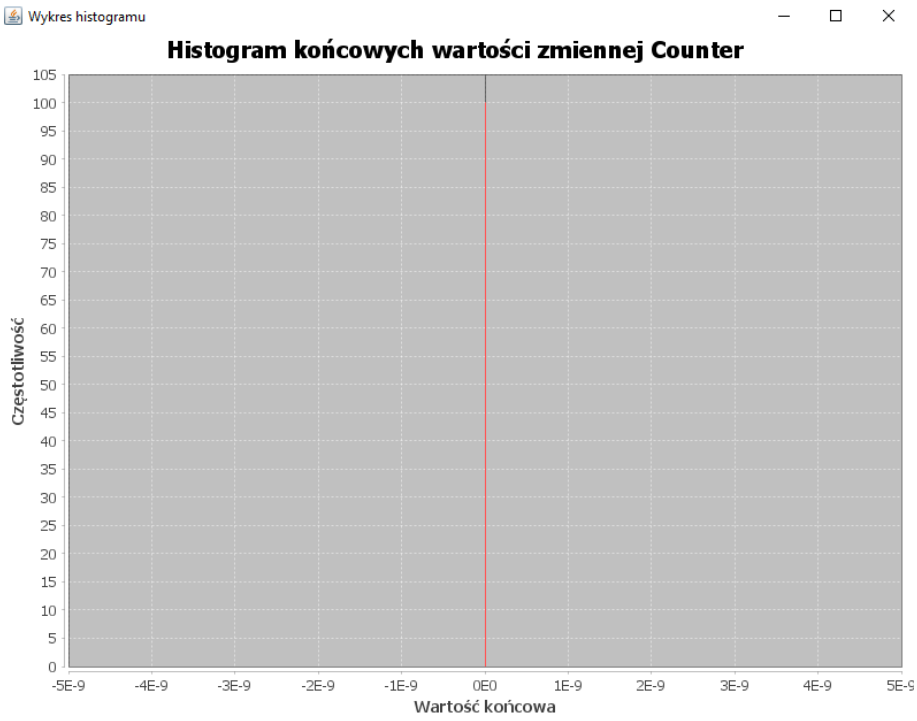
```
                isIncrementTurn = true;
            }
        });

        incrementThread.start();
        decrementThread.start();

        incrementThread.join();
        decrementThread.join();

        System.out.println("Final value of counter: " + counter);
        return counter;
    }
}
```

5 Synchronizacja z wykorzystaniem mechanizmów wbudowanych

W poniższym kodzie zaimplementowano synchronizację dwóch wątków (inkrementującego i dekrementującego zmienną `counter`) przy użyciu wbudowanych mechanizmów synchronizacji w Javie, takich jak `synchronized()` oraz `wait()` i `notify()`. Mechanizmy te pozwalają na bezpieczne i przewidywalne przełączanie się wątków między sobą, eliminując problem warunków wyścigu.

```
public class Main {
    private static int counter = 0;
    private static boolean isIncrementTurn = true;
    private static final Object lock = new Object();

    public static void main(String[] args) throws InterruptedException {
        int[] results = new int[100];

        for (int i = 0; i < 100; i++) {
            results[i] = simulateOperations();
        }
        HistogramPlotter.plotHistogram(results);
    }

    public static int simulateOperations() throws InterruptedException {
        counter = 0;
        isIncrementTurn = true;

        Thread incrementThread = new Thread(() -> {
            for (int i = 0; i < 10000; i++) {
                synchronized (lock) {

                    while (!isIncrementTurn) {
                        try {
                            lock.wait();
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }
                    }
                    counter++;
                    isIncrementTurn = false;
                    lock.notify();
                }
            }
        });

        Thread decrementThread = new Thread(() -> {
```

```
            for (int i = 0; i < 10000; i++) {
                synchronized (lock) {

                    while (isIncrementTurn) {
                        try {
                            lock.wait();
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }
                    }
                    counter--;
                    isIncrementTurn = true;
                    lock.notify();
                }
            }
        });

        incrementThread.start();
        decrementThread.start();

        incrementThread.join();
        decrementThread.join();

        System.out.println("Final value of counter: " + counter);
        return counter;
    }
}
```



Rysunek 4: Histogram końcowych wartości zmiennej `counter` po 100 iteracjach z użyciem mechanizmów wbudowanych

W rezultacie, dzięki mechanizmom synchronizacji, wynik końcowy zmiennej `counter` zawsze wynosi 0. Dzieje się tak, ponieważ liczba operacji inkrementacji i dekrementacji jest taka sama, a synchronizacja gwarantuje, że operacje te nie zostaną przypadkowo pominięte przez równoczesny dostęp wątków.

6 Bibliografia

1. Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, Doug Lea. ***Java Concurrency in Practice***. Addison-Wesley Professional, 2006. ISBN: 978-0321349606.
2. Oracle. ***The Java Tutorials - Concurrency***. Oracle, 2023.
3. David Gilbert. ***JFreeChart Developer Guide***. JFree.org, 2023.
4. Hans Boehm. ***"Threads Cannot Be Implemented As a Library"***. ACM SIGPLAN Notices, Vol. 40, No. 6, 2005, pp. 261–268.