Łukasz Oprych Gr. 5 Lab, 410687	Programowanie Równoległe	Temat: Java

Cel ćwiczenia:

Celem ćwiczenia było zapoznanie się z tworzeniem programów wielowątkowych oraz opanowanie podstawowych metod synchronizacji w języku Java.

Przebieg ćwiczenia:

Wariant nr 1

Po pobraniu plików oraz przygotowaniu środowiska zgodnie z poleceniem prowadzącego wykonano pierwszy wariant zadania początkowo poprzez stworzenie nowej klasy Watek w pliku Histogram_test, która dziedziczy po klasie Thread.

```
class Watek extends Thread{

        public void run (){
        obraz.calculate_histogram_parallel(znak_index);
        obraz.print_histogram_parallel(znak_index);
        }

        public Watek (Obraz obraz, int znak_index){
        this.obraz = obraz;
        this.znak_index = znak_index;
        }
        private Obraz obraz;
        private int znak_index;
}
```

W klasie dodano konstruktor, gdzie przekazujemy obraz i indeks znaku, jest też wywoływana funkcja run korzystająca z metod klasy Obraz, obliczających ilość powtórzeń danego znaku w histogramie oraz wyświetlanie.

Funkcja main klasy Histogram_test:

```
public static void main(String[] args) {

    Scanner scanner = new Scanner(System.in);

    System.out.println("Set image size: n (#rows), m(#kolumns)");
    int n = scanner.nextInt();
    int m = scanner.nextInt();
    Obraz obraz_1 = new Obraz(n, m);

    obraz_1.calculate_histogram();
    obraz_1.print_histogram();

    System.out.println("########### Wersja Równoległa #########");

     int num_threads = 94;
     Watek[] NewThr = new Watek[num_threads];

     for (int i = 0; i < num_threads; i++) {
         (NewThr[i] = new Watek(obraz_1, i)).start(); //stworzenie i uruchomienie watku
     }

     for (int i = 0; i < num_threads; i++) {
         try {
            NewThr[i].join();
         } catch (InterruptedException e) {}
     }
    obraz_1.check_histograms();
}
```

Stworzenie w klasie Obraz tablicy histogram_parallel do przechowywania wyników obliczeń równoległych, początkowo jest wypełniona wartością 0 w metodzie clear_histogram_parallel.

```
histogram_parallel = new int[94];
```

```
public void clear_histogram_parallel(){
    for(int i=0;i<94;i++) histogram_parallel[i]=0;
}
```

Aby obliczyć ilość powtórzonych znaków wykorzystano metodę calculate_histogram_parallel, oraz aby sprawdzić czy liczba wystąpień znaku danego wątku jest zgodna w wersji sekwencyjnej oraz równoległej metodę check_histograms.

```
public void calculate_histogram_parallel(int znak_index) {
    for(int i=0;i<size_n;i++) {
        for(int j=0;j<size_m;j++) {
            if(tab[i][j] == tab_symb[znak_index]){
                    histogram_parallel[znak_index]++;
            }
        }
    }
}
public void check_histograms () {

    for(int i=0; i<94; i++){
    if (histogram[i] != histogram_parallel[i]){
      System.out.println("Wartość wersji sekwencyjnej nie zgadza się z wersją równoległą");
    return;
    }
    }
      System.out.println("Wszystkie wartości w obu wersjach są zgodne");
}
```

Stworzono metodę print_histogram_parallel do wyświetlania znaku „=” przy powtórzeniu się znaku w danym wątku. Jak widać, użyto synchronized, które odpowiada za zabezpieczenie sekcji krytycznej, aby tylko jeden wątek wykonywał ten fragment kodu do czasu opuszczenia przez niego tej sekcji.

```
public synchronized void print_histogram_parallel(int znak_index) {

 System.out.print("Wątek " + znak_index + " : " + tab_symb[znak_index] + " ");
    for(int i=0; i<  histogram_parallel[znak_index]; i++){
     System.out.print("=");
    }
    System.out.print("\n");

}
```

Stworzenie i uruchomienie wątków dla pierwszego przypadku:

```
for (int i = 0; i < num_threads; i++) {
    (NewThr[i] = new Watek(obraz_1, i)).start();
}
```

Wynik dla tablicy o wymiarze 10x10:

```
Wątek 25 : : ===
Wątek 26 : ; =
Wątek 30 : ? =
Wątek 31 : @ ==
Wątek 24 : 9
```

```
$ 2
% 1
& 0
' 1
( 0
) 1
* 3
```

```
Wątek 93 : ~ =
Wątek 92 : } ==
Wątek 86 : w
Wątek 89 : z =
Wątek 88 : y =
Wszystkie wartości w obu wersjach są zgodne
```

Wariant nr 2

W tym przypadku dochodzi do dekompozycji blokowej.

Utworzenie klasy wątek, która implementuje interfejs Runnable, oraz zmodyfikowanie jej pod dany wariant.

```
class Watek implements Runnable{
        public void run (){

        obraz.calculate_histogram_parallel(start_index, end_index);
        obraz.print_histogram_parallel(start_index, end_index, watek_index);
        }

        public Watek (Obraz obraz, int watek_index, int start_index, int end_index){
        this.obraz = obraz;
        this.end_index = end_index;
        this.start_index = start_index;
        this.watek_index = watek_index;
        }
        private Obraz obraz;
        private int start_index;
        private int end_index;
        private int watek_index;
}
```

Modyfikacja funkcji main:

```
System.out.println("Set number of threads");
 int num_threads = scanner.nextInt();

 Thread[] NewThr = new Thread[num_threads];
 int numSignsPerThread = 94 / num_threads;
 int divisionRest = 94 % num_threads;
 int startIndex = 0;
 int endIndex = numSignsPerThread;
 if (divisionRest > 0) {
        divisionRest--;
        endIndex++;
 }
```

Zmienna divisionRest gwarantuje, że póki będzie wychodzić więcej niż 0, to te dane będą przekazywane do pierwszych wątków. (Żeby ostatni watek nie miał dużo danych, w celu zapewnienia wydajności).

Utworzenie funkcji calculate_histogram_parallel, oraz funkcji print_histogram_parallel.

```
public void calculate_histogram_parallel(int start_index, int end_index) {
    for(int i=0;i<size_n;i++) {
        for(int j=0;j<size_m;j++) {
            for(int k=start_index; k<end_index; k++){
                    if(tab[i][j] == tab_symb[k]){
                            histogram_parallel[k]++;
                    }
            }
        }
    }
}

public synchronized void print_histogram_parallel(int start_index, int end_index, int watek_index) {
 for(int j=start_index; j<end_index; j++) {
 System.out.print("Wątek " + watek_index + ": " + tab_symb[j] + " ");
    for(int i=0; i<histogram_parallel[j]; i++){
     System.out.print("=");
    }
    System.out.print("\n");
 }
}
```

Wynik dla tablicy 20x20 oraz zastosowanych czterech wątków:

```
n 2
o 3
p 6
q 5
r 6
s 4
t 6
u 7
v 7
w 10
x 6
y 6
z 4
{ 3
```

```
########### Wersja Równoległa #########
Set number of threads
4
Wątek 0: ! ====
Wątek 0: " ===
Wątek 0: # ====
Wątek 0: $ ==
Wątek 0: % ====
Wątek 0: & =========
Wątek 0: ' =====
Wątek 0: ( ===
Wątek 0: ) =====
Wątek 0: * =====
```

```
Wątek 1: G
Wątek 1: H
Wątek 1: I ====
Wątek 1: J =====
Wątek 1: K ====
Wątek 1: L =
Wątek 1: M ========
Wątek 1: N ====
Wątek 1: O ===
Wątek 1: P =======
Wszystkie wartości w obu wersjach są zgodne
loprych@loprych-VirtualBox:~/Desktop/PR_lab/lab_6/2$
```

Wnioski:

W Javie istnieją dwa główne podejścia do tworzenia nowych wątków:

- Poprzez utworzenie własnej klasy, która dziedziczy po klasie Thread. Możemy stworzyć obiekt tej klasy i rozpocząć wątek. Użycie podklasy Thread zajmuje więcej pamięci i nie pozwala na rozszerzanie innej klasy, ponieważ rozszerza już Thread. Pozwala jednak startować bez tworzenia nowej instancji Thread.

- Poprzez utworzenie klasy implementującej interfejs Runnable. Aby uruchomić wątek, musimy przekazać instancję tej klasy jako argument do konstruktora nowego obiektu klasy Thread. Implementacja interfejsu Runnable zajmuje mniej pamięci i daje więcej możliwości, pozwala aby nasza klasa rozszerzała jeszcze inne klasy. Do uruchomienia wątku potrzebuje jednak instancji Thread.

Przy tworzeniu programu, trzeba rozważyć czy dla naszego programu bardziej ważne jest rozszerzanie innych klas czy prostota uruchamiania wątków.

Oznaczenie metody jako synchronized pozwala na zapewnienie bezpieczeństwa działania obiektu pod kątem spójności operacji w przypadku wielowątkowym.