Programowanie Równoległe	Kierunek: Informatyka techniczna	Grupa: 9
Imię i nazwisko: Jakub Świerczyński	Temat: Lab6, lab7	Termin oddania: 25.11.2024

Cel laboratorium

Opanowanie tworzenia wątków oraz metod synchronizacji w Javie.

Realizacja

Wariant 1

1. Modyfikacja klasy Obraz

Dodano nową tablicę hist_parallel do przechowywania histogramu obliczanego równolegle.

Dodano metodę clear_hist_parallel() do czyszczenia tej tablicy.

Dodano metodę calculate_histogram_parallel(char znak), która oblicza liczbę wystąpień danego znaku w obrazie.

Dodano metodę print_histogram_parallel(char znak), która wyświetla licznik wystąpień danego znaku w formie graficznej.

Dodano metodę compare_histograms(), która porównuje histogram sekwencyjny z histogramem równoległym.

```
private int[] hist_parallel;
```

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

```
public void calculate_histogram_parallel(char znak) {
    int count = 0;
    for(int i=0; i<size_n; i++) {
       for(int j=0; j<size_m; j++) {
          if(tab[i][j] == znak) {
             count++;
          }
       }
    }
    int index = (int)znak - 33;
    hist_parallel[index] = count;
}
```

```
public synchronized void print_histogram_parallel(char znak) {
    int index = (int)znak - 33;
    int count = hist_parallel[index];
    System.out.print(Thread.currentThread().getName() + ": " + znak + " ");
    for(int i=0; i<count; i++) {
       System.out.print("=");
    }
    System.out.println();
}
```

2. Modyfikacja klasy Histogram_test

W funkcji main tworzymy wątki dla każdego znaku z tablicy tab_symb.

Uruchamiamy wątki i czekamy na ich zakończenie.

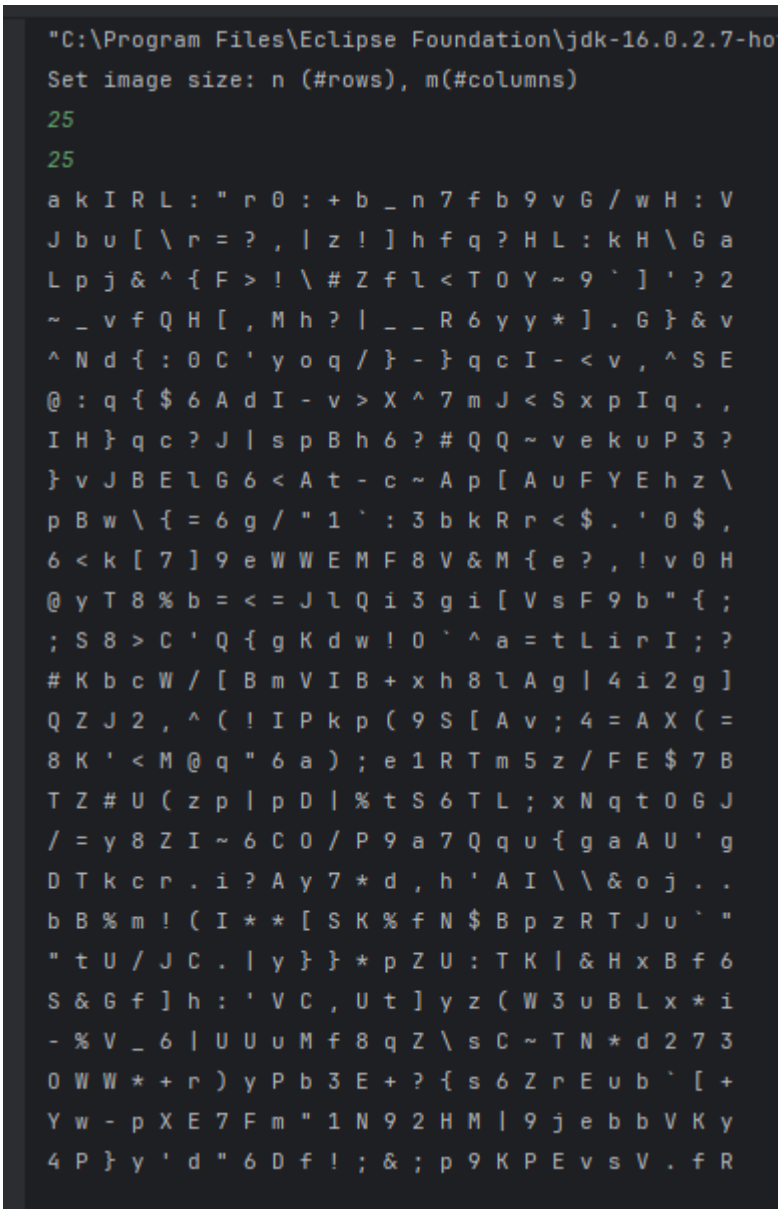
Porównujemy histogramy.

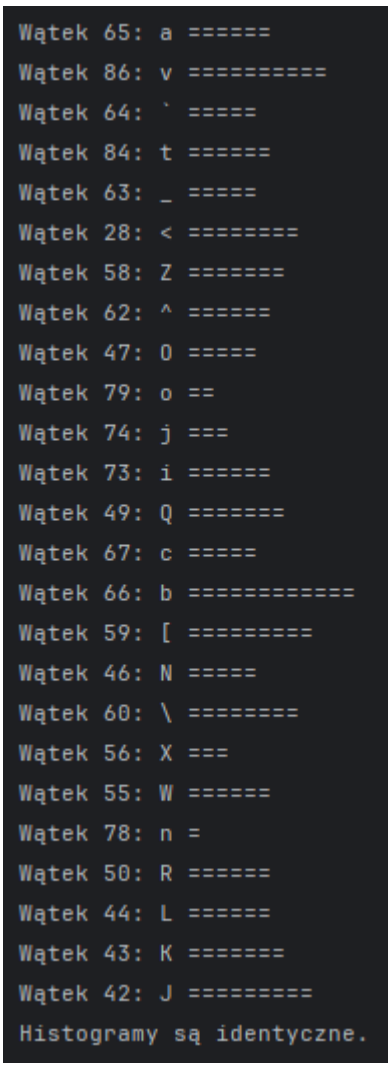
```
import java.util.Scanner;

class Histogram_test {

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

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

        // Obliczanie i drukowanie histogramu sekwencyjnie
        obraz_1.calculate_histogram();
        obraz_1.print_histogram();

        // Tworzenie wątków dla każdego znaku
        CharacterThread[] watki = new CharacterThread[94];

        for (int i = 0; i < 94; i++) {
            watki[i] = new CharacterThread(obraz_1.tab_symb[i], obraz_1);
            watki[i].setName("Wątek " + (i + 1));
            watki[i].start();
        }

        // Oczekiwanie na zakończenie wszystkich wątków
        for (int i = 0; i < 94; i++) {
            try {
                watki[i].join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        // Porównanie histogramów
        obraz_1.compare_histograms();

    }

}
```

Wyniki

Każdy wątek wyświetla licznik wystąpień swojego znaku w formie graficznej.

Histogramy sekwencyjny i równoległy są identyczne, co potwierdza poprawność implementacji.





Wariant 2

1. Modyfikacja klasy Obraz

Dodano metodę calculate_histogram_parallel_range(int startIndex, int endIndex), która oblicza histogram dla zakresu znaków.

Dodano metodę print_histogram_parallel_range(int startIndex, int endIndex), która wyświetla fragment histogramu dla danego zakresu.

```
public void calculate_histogram_parallel_range(int startIndex, int endIndex) {
    for(int i=0; i<size_n; i++) {
       for(int j=0; j<size_m; j++) {
          char znak = tab[i][j];
          int index = (int)znak - 33;
          if(index >= startIndex && index < endIndex) {
             synchronized(this) {
                hist_parallel[index]++;
             }
          }
       }
    }
}
public synchronized void print_histogram_parallel_range(int startIndex, int endIndex) {
    for(int k=startIndex; k<endIndex; k++) {
       char znak = tab_symb[k];
       int count = hist_parallel[k];
       System.out.print(Thread.currentThread().getName() + ": " + znak + " ");
       for(int i=0; i<count; i++) {
          System.out.print("=");
       }
       System.out.println();
    }
}
```

2. Modyfikacja klasy CharacterThread

Klasa CharacterThread implementuje interfejs Runnable.

Konstruktor przyjmuje początkowy i końcowy indeks zakresu oraz obiekt Obraz.

Metoda run() wywołuje metody obliczające i drukujące fragment histogramu dla danego zakresu.

```
class CharacterThread implements Runnable {
    private int startIndex;
    private int endIndex;
    private Obraz obraz;

    public CharacterThread(int startIndex, int endIndex, Obraz obraz) {
        this.startIndex = startIndex;
        this.endIndex = endIndex;
        this.obraz = obraz;
    }

    @Override
    public void run() {
        obraz.calculate_histogram_parallel_range(startIndex, endIndex);
        obraz.print_histogram_parallel_range(startIndex, endIndex);
    }
}
```

3. Modyfikacja klasy Histogram_test

W funkcji main pytamy użytkownika o liczbę wątków.

Obliczamy zakres znaków dla każdego wątku.

Tworzymy i uruchamiamy wątki odpowiedzialne za swoje zakresy.

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

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

```
for (int i = 0; i < num_threads; i++) {
    int startIndex = i * zakres;
    int endIndex = (i == num_threads - 1) ? 94 : startIndex + zakres;

    CharacterThread watekZakres = new CharacterThread(startIndex, endIndex, obraz_1);
    watki[i] = new Thread(watekZakres);
    watki[i].setName("Wątek " + (i + 1));
    watki[i].start();
}

// Oczekiwanie na zakończenie wszystkich wątków
for (int i = 0; i < num_threads; i++) {
    try {
        watki[i].join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

// Porównanie histogramów
obraz_1.compare_histograms();
```

Wyniki

```
15
15
+ a Y n x H T > % l l x 0 { N
c A $ S ] 9 & Y _ ) J X 6 2 t
E X y A x ( q + v f ( 3 f W h
A ! J 9 / U N v / ' R A E 3 ?
T 3 H Q + _ [ T p 3 F O ( 0 &
J [ U * ' 7 U K 0 ' 3 C l " P
= ^ q ; 3 G t A ( ) { + 2 p Z
s } c [ " N 5 H E S f 0 . ~ y
{ { $ u v 0 y ! 9 " p q | ] !
= H R 1 L G S ) I u . # $ c T
O N F R 0 ' 9 } 1 B ) Z ( < G
" l ^ ` ; 8 = * E f U e ~ ^ K
f = I s ~ $ c H L 1 v 9 @ . H
[ Z C ? 3 c y y y . g > Y ; 8
J M _ p n l u q l 8 E W > ( y
```

```
Wątek 9: l ======
Wątek 9: m
Wątek 9: n ==
Wątek 9: o
Wątek 9: p ====
Wątek 9: q ====
Wątek 4: < =
Wątek 4: = ====
Wątek 4: > ===
Wątek 4: ? ==
Wątek 4: @ =
Wątek 4: A =====
Wątek 4: B =
Wątek 4: C ==
Wątek 4: D
Histogramy są identyczne.
```

Każdy wątek wyświetla licznik wystąpień swoich znaków w formie graficznej.

Histogramy sekwencyjny i równoległy są identyczne.

Wnisoki

Poprzez zastosowanie wątków w Javie, możliwe jest zrównoleglenie obliczeń i przyspieszenie wykonywania programów.

Synchronizacja jest kluczowa w programowaniu wielowątkowym, aby zapewnić poprawność danych współdzielonych między wątkami.

Dekompozycja zadania na wątki powinna być przemyślana pod kątem efektywności i wykorzystania zasobów.

Laboratorium 7

Cel laboratorium

Celem laboratorium było nabycie umiejętności pisania programów w języku Java z wykorzystaniem puli wątków.

Zadanie 1

Obliczanie całki metodą trapezów

Implementacja sekwencyjna

Na początek mieliśmy napisać sekwencyjny program obliczania całki z zadanej funkcji metodą trapezów, korzystając z dostarczonej klasy Calka_callable. Klasa ta oblicza całkę w zadanym przedziale z dokładnością określoną przez parametr dx.

Klasa Calka_callable

Pierwotnie klasa Calka_callable wyglądała następująco:ejętności pisania programów w języku Java z wykorzystaniem puli wątków

Aby klasa Calka_callable mogła być wykorzystana jako zadanie dla puli wątków, zaimplementowaliśmy interfejs Callable<Double>:

```
public class Calka_callable implements Callable<Double>{
```

```
@Override
public Double call() {
    double calka = 0.0;
    int N = (int) Math.ceil((xk-xp) / dx);
    double actualDx = (xk - xp) / N;

    for (int i = 0; i < N; i++) {
        double x1 = xp + actualDx * i;
        double x2 = x1 + actualDx;
        calka += ((getFunction(x1) + getFunction(x2)) / 2) * actualDx;
    }
    System.out.println("Watek " + Thread.currentThread().getName() +
            ": " + calka);
    return calka;
}
```

W metodzie main dokonaliśmy następujących kroków:

Ustawienie parametrów całkowania:

Przedział całkowania: start = 0.0, end = Math.PI.

Dokładność całkowania: dx = 0.002.

Liczba wątków: nthreads = 10.

Liczba zadań: ntasks = 40.

Utworzenie puli wątków:

```
ExecutorService executor = Executors.newFixedThreadPool(nthreads);
```

Podział przedziału całkowania na podprzedziały:

```
double actualDx = (end - start) / ntasks;
```

Tworzenie i przekazywanie zadań do wykonania:

```
for (int i = 0; i < ntasks; i++) {
    double x1 = start + i * actualDx;
    double x2 = x1 + actualDx;

    Calka_callable task = new Calka_callable(x1, x2, dx);

    Future<Double> result = executor.submit(task);

    results.add(result);
}
```

Odbieranie wyników i sumowanie całki:

```
double calkaResult = 0;

for (Future<Double> result : results) {
    try {
        calkaResult += result.get();
    } catch (InterruptedException | ExecutionException e) {
        e.printStackTrace();
    }
}
executor.shutdown();

System.out.println("Calka result: "+ calkaResult);
```

Wynik

Uruchomienie programu daje wynik zbliżony do dokładnej wartości całki z sin(x) w przedziale od 0 do π, czyli 2.

```
Watek pool-1-thread-2: 0.009228990172934354
Watek pool-1-thread-1: 0.003082665276484703
Watek pool-1-thread-4: 0.021313397255033433
Watek pool-1-thread-9: 0.04861101315734961
Watek pool-1-thread-10: 0.05329916728972368
Calka result: 1.9999993574475892
```

Zadanie 2

Sortowanie przez scalanie z wykorzystaniem ForkJoinPool

W tym zadaniu mieliśmy zaimplementować równoległe sortowanie przez scalanie z wykorzystaniem klasy ForkJoinPool.

Klasa DivideTask

Klasa DivideTask dziedziczy po RecursiveTask<int[]> i reprezentuje zadanie dzielące tablicę na mniejsze części, aż do osiągnięcia bazowego przypadku, w którym tablica jest sortowana sekwencyjnie.

```
public class DivideTask extends RecursiveTask<int[]> {

    private int[] arrayToDivide;

    public DivideTask(int[] arrayToDivide) {
        this.arrayToDivide = arrayToDivide;
    }
```

Główna klasa programu MergeSortForkJoin

```
public class MergeSortForkJoin {

    public static void main(String[] args) {
        int size = 20;
        int[] array = generateRandomArray(size);

        System.out.println("Tablica przed sortowaniem:");
        System.out.println(Arrays.toString(array));

        ForkJoinPool pool = new ForkJoinPool();

        DivideTask mainTask = new DivideTask(array);

        int[] sortedArray = pool.invoke(mainTask);

        if (isSorted(sortedArray)) {
            System.out.println("Tablica jest poprawnie posortowana.");
        } else {
            System.out.println("Błąd w sortowaniu.");
        }

        System.out.println("Tablica po sortowaniu:");
        System.out.println(Arrays.toString(sortedArray));
    }

    private static int[] generateRandomArray(int size) {
        Random rand = new Random();
        int[] result = new int[size];

        for (int i = 0; i < size; i++) {
            result[i] = rand.nextInt(100);
        }

        return result;
    }

    private static boolean isSorted(int[] array) {
        for (int i = 1; i < array.length; i++) {
            if (array[i] < array[i - 1]) {
                return false;
            }
        }
        return true;
    }
}
```

Wynik

```
Tablica przed sortowaniem:
[37, 16, 83, 82, 35, 13, 44, 19, 67, 12, 0, 15, 9, 91, 94, 95, 79, 17, 55, 86]
Tablica jest poprawnie posortowana.
Tablica po sortowaniu:
[0, 9, 12, 13, 15, 16, 17, 19, 35, 37, 44, 55, 67, 79, 82, 83, 86, 91, 94, 95]
```

Wnioski

Pule wątków w Javie są potężnym narzędziem umożliwiającym równoległe wykonywanie zadań, co może znacząco przyspieszyć wykonywanie programów, zwłaszcza na wielordzeniowych procesorach.

Interfejsy Callable i Runnable pozwalają na definiowanie zadań, które mogą być wykonywane przez wątki.

Klasa ForkJoinPool jest szczególnie użyteczna w przypadku algorytmów rekurencyjnych, takich jak sortowanie przez scalanie, gdzie zadania mogą być dzielone na mniejsze podzadania i wykonywane równolegle.

Synchronizacja i kontrola wątków są kluczowe w programowaniu równoległym, aby zapewnić poprawność i spójność danych.