Programowanie równoległe	Kierunek: Informatyka Techniczna	Grupa: 9
Imię i nazwisko: Karolina Starzec	Temat: LAB 6 I 7	Termin oddania: 24.11.2024

LAB 6

1. Cel

Opanowanie podstaw tworzenia wątków w Javie oraz opanowanie podstawowych metod synchronizacji w Javie

2. Realizacja zajęć

Realizując zajęcia skopiowałam i rozpakowałam potrzebne pliki ze strony przedmiotu oraz umieściłam je w odpowiednim podkatalogu. Następnie sprawdziłam działanie kodu Histogram_test.java oraz Obraz.java. Kolejnym krokiem było rozszerzenie i modyfikacja kodu, w taki sposób, żeby obliczanie histogramu odbywało się w sposób równoległy, przy użyciu wątków Javy. Dopisanie odpowiednich metod w pliku Obraz.java:

update_histogram to synchronizowana metoda, która aktualizuje histogram, zwiększając wartość dla indeksu odpowiadającego podanemu symbolowi ASCII. Indeks jest obliczany jako różnica między kodem ASCII symbolu a wartością 33.

getSymbolCount to synchronizowana metoda zwracająca liczbę wystąpień określonego symbolu ASCII w histogramie. Indeks symbolu w histogramie obliczany jest w ten sam sposób jak w metodzie update_histogram.

print_symbol_count to synchronizowana metoda, która wyświetla licznik wystąpień symbolu w formie graficznej (symbol oraz ilość przedstawiona jako rząd znaków '='). Dodatkowo wyświetlany jest numer wątku, który wykonuje tę operację.

```
// Synchronizowana metoda do aktualizacji histogramu dla danego symbolu
public synchronized void update_histogram(char symbol) {
    int index = (int) symbol - 33; // Obliczanie indeksu w histogramie
    histogram[index]++;
}

// Metoda do pobierania liczby wystąpień danego symbolu z histogramu
public synchronized int getSymbolCount(char symbol) {
    int index = (int) symbol - 33;
    return histogram[index];
}
// Synchronizowana metoda do wyświetlania liczby wystąpień symbolu w formie
graficznej
public synchronized void print_symbol_count(char symbol, int count, int
threadNumber) {
    System.out.print("Wątek " + threadNumber + ": " + symbol + " ");
    for (int i = 0; i < count; i++) {
        System.out.print("=");
    }
    System.out.println();
}
```

Następnie stworzyłam dwie kolejne klasy WatekWariant1 i WatekWariant2.

Wariant 1:

Jest to metoda obliczania histogramu, gdzie każdy wątek odpowiada za jeden symbol ASCII, a logika wątków jest zaimplementowana przez klasę dziedziczącą po Thread. Dzięki temu każdy z 94 wątków może być uruchamiany bezpośrednio za pomocą metody start. Wątki liczą wystąpienia przypisanych symboli w tablicy znaków, a następnie w sposób synchronizowany aktualizują globalny histogram i wypisują wyniki, zapewniając bezpieczeństwo współbieżnych operacji.

```
class WatekWariant1 extends Thread {
    private char symbol; // Przypisany symbol dla tego wątku
    private Obraz obraz;
    private int threadNumber; // Numer wątku

    public WatekWariant1(char symbol, Obraz obraz, int threadNumber) {
        this.symbol = symbol;
        this.obraz = obraz;
        this.threadNumber = threadNumber;
    }

    @Override
    public void run() {
        char[][] tab = obraz.getTab(); // Pobieramy tablicę znaków z
obiektu Obraz
        int count = 0;

        // Liczenie wystąpień przypisanego symbolu w tablicy obrazu
        for (int i = 0; i < tab.length; i++) {
            for (int j = 0; j < tab[i].length; j++) {
                if (tab[i][j] == symbol) {
                    count++;
                }
            }
        }

        // Aktualizacja histogramu w obiekcie klasy Obraz
        for (int i = 0; i < count; i++) {
            obraz.update_histogram(symbol);
        }

        // Wyświetlanie liczby wystąpień symbolu w formie graficznej
        obraz.print_symbol_count(symbol, count, threadNumber);
    }
}
```

Przykład wyników dla Wariant 1:

```
Podaj rozmiar obrazu: n (#wierszy), m (#kolumn):
10
10
S I F s ' @ [ k s t
Y d O Q / c " ` j :
. d $ E : z 4 @ _ @
n J & : v l ] D @ k
X O W 8 k d z b F ;
` l t q W k l ( v w
C 8 ? V " 6 2 > < /
C 2 / N g 4 , , h F
B ] 0 ) x Y v B / J
/ 5 < M i U : k + w


Wybierz wariant (1 lub 2):
1
Wątek 1: !
Wątek 94: ~
Wątek 93: }
Wątek 92: |
Wątek 91: {
Wątek 90: z ==
Wątek 89: y
Wątek 88: x =
```

```
Wątek 16: 0 =
Wątek 15: / =====
Wątek 14: . =
Wątek 13: -
Wątek 12: , ==
Wątek 11: + =
Wątek 10: *
Wątek 9: ) =
Wątek 8: ( =
Wątek 7: ' =
Wątek 6: & =
Wątek 5: %
Wątek 4: $ =
Wątek 3: #
Wątek 2: " ==
Histogram dla wariantu 1 został wyświetlony.
```

Wariant 2

Jest to metoda obliczania histogramu, w której praca jest podzielona między wątki na zakresy symboli ASCII. Wykorzystuje interfejs Runnable, co pozwala na elastyczną implementację wątków obsługiwanych przez klasę Thread. Każdy wątek liczy wystąpienia symboli ze swojego zakresu, zapisuje je w lokalnym histogramie, a następnie, za pomocą mechanizmu synchronized, aktualizuje globalny histogram, unikając konfliktów w dostępie do wspólnych danych. Synchronizacja zapewnia również, że wyniki wypisywane na ekran są czytelne i nie nakładają się na siebie, gwarantując bezpieczeństwo współbieżnych operacji.

```
class WatekWariant2 implements Runnable {
    private final Obraz obraz;
    private final int startSymbol;
    private final int endSymbol;
    private final int threadNumber;

    public WatekWariant2(int startSymbol, int endSymbol, Obraz obraz, int
threadNumber) {
        this.startSymbol = startSymbol;
        this.endSymbol = endSymbol;
        this.obraz = obraz;
        this.threadNumber = threadNumber;
    }
  @Override
    public void run() {
        char[][] tab = obraz.getTab(); // Pobranie obrazu
        char[] tabSymb = obraz.getTabSymb(); // Pobranie symboli ASCII
        int[] localHistogram = new int[94]; // Lokalny histogram wątku

        // Liczenie wystąpień dla przypisanych znaków
        for (int k = startSymbol; k < endSymbol; k++) {
            char symbol = tabSymb[k];
            int localCount = 0;
            for (int i = 0; i < tab.length; i++) {
                for (int j = 0; j < tab[i].length; j++) {
                    if (tab[i][j] == symbol) {
                        localCount++;
                    }
                }
            }
            localHistogram[k] = localCount;

            // Synchronizowane wyświetlanie wyników
            synchronized (obraz) {
                obraz.print_symbol_count(symbol, localCount, threadNumber);
            }
        }

        // Aktualizacja globalnego histogramu
        for (int k = startSymbol; k < endSymbol; k++) {
            obraz.update_histogram((char) (k + 33)); // Aktualizacja
globalna
        }
    }
}
```

Przykład wyników dla Wariant 2:

```
Podaj liczbę wątków dla wariantu 2:
4
Wątek 1: !
Wątek 4: f
Wątek 4: g =
Wątek 2: 8 ==
Wątek 2: 9
Wątek 2: : ====
Wątek 3: O ==
Wątek 3: P
Wątek 2: ; =
Wątek 2: < ==
Wątek 4: h =
Wątek 4: i =
Wątek 1: " ==
Wątek 1: #
Wątek 1: $ =
Wątek 4: j =
Wątek 4: k =====
Wątek 4: l ===
Wątek 4: m
Wątek 2: =
Wątek 3: Q =
Wątek 3: R
```

```
Wątek 1: / =====
Wątek 1: 0 =
Wątek 1: 1
Wątek 1: 2 ==
Wątek 1: 3
Wątek 1: 4 ==
Wątek 1: 5 =
Wątek 1: 6 =
Wątek 1: 7
Histogram dla wariantu 2 został wyświetlony.
```

Na sam koniec wyświetlamy jeszcze porównanie obu histogramów za pomocą funkcji compareHistograms. Sprawdza zgodność wyników obu wariantów, iterując przez indeksy i porównując wartości dla każdego symbolu ASCII (33–126). W przypadku różnic wyświetla szczegóły: symbol, liczby wystąpień w obu histogramach i różnicę. Na koniec zwraca true, jeśli histogramy są identyczne, lub false, jeśli występują różnice.

```java
public boolean compareHistograms(int[] histParallel) {
    boolean match = true;
    char[] tabSymb = this.getTabSymb();

    System.out.println("\nPorównanie histogramów:");
    System.out.println("Symbol | Sekwencyjny | Równoległy | Różnica");
    System.out.println("-------|-------------|------------|--------");

    for (int i = 0; i < 94; i++) {
        int diff = this.histogram[i] - histParallel[i];
        System.out.printf("   %c   |      %3d     |      %3d     |    %3d\n",
                tabSymb[i], this.histogram[i], histParallel[i], diff);
        if (diff != 0) {
            match = false;
        }
    }

    if (match) {
        System.out.println("\nWszystkie wartości pasują.");
    } else {
        System.out.println("\nWykryto różnice w histogramach.");
    }
    return match;
}
```

Przykład wyników porównania:

```
Porównanie histogramów:
Symbol | Sekwencyjny | Równoległy | Różnica
-------|-------------|------------|--------
   !   |      1      |      1     |     0
   "   |      1      |      1     |     0
   #   |      1      |      1     |     0
   $   |      1      |      1     |     0
   %   |      1      |      1     |     0
   &   |      1      |      1     |     0
   '   |      1      |      1     |     0
   (   |      1      |      1     |     0
   )   |      1      |      1     |     0
   *   |      1      |      1     |     0
   +   |      1      |      1     |     0
   ,   |      1      |      1     |     0
   -   |      1      |      1     |     0
   .   |      1      |      1     |     0
   /   |      1      |      1     |     0
   0   |      1      |      1     |     0
   1   |      1      |      1     |     0
   2   |      1      |      1     |     0
   3   |      1      |      1     |     0
   4   |      1      |      1     |     0
```

```
Wszystkie wartości pasują.

Histogramy są identyczne.
```

3. Wnioski

W Wariancie 1 synchronizacja nie jest wymagana, ponieważ każdy wątek zajmuje się jednym konkretnym symbolem ASCII, eliminując rywalizację o wspólne dane. Każdy wątek ma przypisany inny fragment pracy, co zapewnia naturalną separację zadań i brak konieczności ochrony współdzielonych zasobów.

W Wariancie 2 synchronizacja jest konieczna, ponieważ wiele wątków pracuje równolegle na tym samym globalnym histogramie, co wymaga ochrony przed równoczesnym dostępem. Mechanizm synchronized zapewnia, że dane są aktualizowane poprawnie i żaden wątek nie nadpisze wyników innego. Synchronizacja jest także używana do wypisywania wyników na ekran w sposób spójny i czytelny.

Liczbę wątków w Wariancie 2 można dostosować do liczby dostępnych rdzeni procesora, dzieląc zakres symboli na odpowiednie bloki. Dzięki temu program lepiej wykorzystuje zasoby sprzętowe, unikając zarówno nadmiarowego obciążenia systemu, jak i niewykorzystania jego potencjału.

Brak różnic w wynikach obu wariantów wynika z deterministycznego charakteru obliczeń oraz poprawnie zaimplementowanego mechanizmu synchronizacji w Wariancie 2. Zarówno w podejściu sekwencyjnym, jak i równoległym, dane wejściowe oraz logika obliczeń są takie same, co gwarantuje spójne rezultaty.

Zastosowanie Thread w Wariancie 1 umożliwia prostą implementację wątków, dzięki której każdy wątek działa jako osobna klasa dziedzicząca po Thread. W Wariancie 2 wykorzystano interfejs Runnable, co zapewnia większą elastyczność, umożliwiając łatwe przekazywanie logiki wątku do obiektów klasy Thread. Jest to bardziej uniwersalne rozwiązanie, które lepiej wpisuje się w złożone projekty.

Podsumowując, Wariant 1 jest prosty, ale nie skalowalny, ponieważ liczba wątków jest stała i zawsze wynosi 94, co może być nieefektywne na systemach o ograniczonej liczbie rdzeni procesora. Wariant 2 jest bardziej elastyczny i praktyczny, ponieważ pozwala na dostosowanie liczby wątków do dostępnych zasobów, co czyni go bardziej efektywnym w systemach wieloprocesorowych. Program skutecznie demonstruje różnice w podejściach do współbieżności, znaczenie synchronizacji i elastyczność w zarządzaniu wątkami.

LAB 7

1. Cel

Nabycie umiejętności pisania programów w języku Java z wykorzystaniem puli wątków.

2. Realizacja zajęć

Zadanie polegało na zmodyfikowaniu programu tak, aby obliczać całkę metodą trapezów z użyciem puli wątków (ExecutorService). Funkcja, którą wykorzystałam na poczet tych zajęć to sin(x). Przedział całkowania [0, PI] dzielony jest na podprzedziały o równej szerokości, niezależne od dokładności dx. Każdy podprzedział stanowi osobne zadanie realizowane przez obiekt klasy Calka_callable, która została dostosowana do interfejsu Callable<Double> poprzez dodanie metody call().

```
public Double call() throws Exception {
    return compute_integral();
}
```

Tutaj w kodzie tworzymy pulę wątków za pomocą ExecutorService z określoną liczbą wątków (NTHREADS), co pozwala na równoczesne wykonywanie zadań i optymalne wykorzystanie zasobów sprzętowych. Lista results przechowuje obiekty typu Future<Double>, które reprezentują wyniki zadań i umożliwiają ich asynchroniczne pobieranie po zakończeniu obliczeń. Dzięki temu program efektywnie zarządza wielowątkowością, wykonując zadania równolegle i zbierając ich wyniki w sposób uporządkowany.

```
ExecutorService executor = Executors.newFixedThreadPool(NTHREADS);
List<Future<Double>> results = new ArrayList<>();
```

Liczba zadań (NSEGMENTS) oraz liczba wątków (NTHREADS) są niezależnymi parametrami, co pozwala na równoważenie obciążenia. Zwykle liczba zadań jest większa od liczby wątków, dzięki czemu procesor może lepiej wykorzystać dostępne zasoby. Zadania są tworzone i przekazywane do puli w jednej pętli, a ich wyniki odbierane w kolejnej za pomocą obiektów Future. Wyniki cząstkowe są sumowane, aby uzyskać końcowy wynik całki.

```
// Tworzenie zadań dla każdego podprzedziału
for (int i = 0; i < NSEGMENTS; i++) {
    double start = a + i * segmentWidth; // Początek przedziału
    double end = start + segmentWidth; // Koniec przedziału
    Calka_callable task = new Calka_callable(start, end, dx); //Tworzenie
    results.add(executor.submit(task)); // Przekazanie zadania do puli
}
double totalIntegral = 0.0;
// Pobieranie wyników
for (Future<Double> result : results) {
    try {
        double partialResult = result.get(); // Pobranie wyniku cząstkowego
        System.out.println("Wynik cząstkowy: " + partialResult);
        totalIntegral += partialResult; // Dodanie wyniku do całkowitej sumy
    } catch (InterruptedException | ExecutionException e) {
        e.printStackTrace();
    }
}
```

Przykładowe wyniki działania programu:

```
Nowa instancja:
Przedział: 0.0-0.3141592653589793
Ilosc trapezów: 315, Dokładność: 9.973310011396168E-4


Nowa instancja:
Przedział: 0.3141592653589793-0.6283185307179586
Ilosc trapezów: 315, Dokładność: 9.973310011396168E-4


Nowa instancja:
Przedział: 0.6283185307179586-0.9424777960769379
Ilosc trapezów: 315, Dokładność: 9.973310011396168E-4


Nowa instancja:
Przedział: 0.9424777960769379-1.2566370614359172
Ilosc trapezów: 315, Dokładność: 9.973310011396168E-4
```

```
Całka cząstkowa o przedziale [0.6283185307179586, 0.9424777960769379]: 0.2212317237447754
Całka cząstkowa o przedziale [0.9424777960769379, 1.2566370614359172]: 0.2787682348106772
Całka cząstkowa o przedziale [0.3141592653589793, 0.6283185307179586]: 0.1420395101466787
Całka cząstkowa o przedziale [0.0, 0.3141592653589793]: 0.04894347964796534
Całka cząstkowa o przedziale [1.2566370614359172, 1.5707963267948966]: 0.3090169687608083
Całka cząstkowa o przedziale [1.5707963267948966, 1.8849555921538759]: 0.3090169687608085
Całka cząstkowa o przedziale [1.8849555921538759, 2.199114857512855]: 0.27876823481067725
Całka cząstkowa o przedziale [2.199114857512855, 2.5132741228718345]: 0.22123172374477532
Całka cząstkowa o przedziale [2.5132741228718345, 2.827433388230814]: 0.14203951014667865
Całka cząstkowa o przedziale [2.827433388230814, 3.141592653589793]: 0.04894347964796534
```

```
Wynik cząstkowy: 0.04894347964796534
Wynik cząstkowy: 0.14203951014667876
Wynik cząstkowy: 0.22123172374477543
Wynik cząstkowy: 0.2787682348106772
Wynik cząstkowy: 0.3090169687608083
Wynik cząstkowy: 0.3090169687608085
Wynik cząstkowy: 0.27876823481067725
Wynik cząstkowy: 0.22123172374477532
Wynik cząstkowy: 0.14203951014667865
Wynik cząstkowy: 0.04894347964796534
Wynik końcowy: 1.9999998342218102
```

Program poprawnie dzieli przedział całkowania na równe podprzedziały i oblicza całki cząstkowe metodą trapezów, zgodnie z funkcją sin(x). Wyniki cząstkowe są zgodne z oczekiwaniami i zostały poprawnie zsumowane, co dało końcowy wynik około 1.9999998342218102, bardzo bliski teoretycznej wartości 2.

3. Wnioski

Analiza wyników wykazała, że program działa poprawnie, a obliczenia metodą trapezów są zgodne z teoretycznymi wartościami. Możliwość dostosowania liczby wątków i zadań pozwoliła na zrozumienie, że optymalna liczba wątków zależy od zasobów sprzętowych, takich jak liczba rdzeni procesora, oraz od charakterystyki zadań. Idealna liczba wątków często jest równa lub nieco większa od liczby rdzeni.

Laboratoria podkreśliły znaczenie równoważenia liczby wątków i zadań w celu maksymalizacji wydajności oraz elastyczności programu. Dzięki zastosowaniu puli wątków program był łatwy do skalowania i dobrze ilustrował praktyczne zastosowanie wielowątkowości w projektach programistycznych.

Kluczowym elementem było zastosowanie ExecutorService, które umożliwiło efektywne zarządzanie równoległym przetwarzaniem zadań. Dzięki funkcjom takim jak Executors.newFixedThreadPool() i executor.submit(task), program podzielił obliczenia na mniejsze zadania wykonywane równolegle, a wyniki zostały poprawnie odebrane i zsumowane za pomocą Future.