Laboratorium nr. 6, laboratorium nr. 7	
Imię i nazwisko: Miłosz Dębowski	Kierunek: Informatyka Techniczna
Numer albumu: 415045	Gr. Lab.: 8

Laboratorium nr. 6

1. Cel:

   - Opanowanie podstaw tworzenia wątków w Javie.
   - Opanowanie podstawowych metod synchronizacji w Javie.

2. Wykonanie i wyniki

   - Pobranie ze strony przedmiotu pliki z klasami: ‘Obraz.java’ oraz ‘Histogram_test.java’.
   - Analiza klasy Obraz
   - Uruchomienie i testy kodu sekwencyjnego
   - Rozszerzenie kodu o obliczenia równoległe - Implementacja Wariantu 1:
     a) Stworzenie kodu w klasie obraz umożliwiającego obliczenie histogramu równoległego

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

     b) Stworzenie klasy dziedziczącej po `Thread` dla obsługi wątków.

```
class WatekSymbol extends Thread {
    private final Obraz obraz;
    private final char symbol;

    public WatekSymbol(Obraz obraz, char symbol) {
        this.obraz = obraz;
        this.symbol = symbol;
    }

    @Override
    public void run() {
```

```
        obraz.calculate_histogram_parallel(symbol);
        obraz.print_histogram_parallel(symbol);
    }
}
```

c) Implementacja wątków:
   - Każdy wątek przetwarza wystąpienia tylko jednego znaku w tablicy.
   - Wątek korzysta z metod klasy ‘Obraz’.

d) Zarządzanie wątkami w ‘main’:
   - Utworzenie wątków dla każdego znaku obecnego w tablicy.
   - Uruchomienie wątków i oczekiwanie na zakończenie ich pracy.

```
System.out.println("Uruchamianie wersji równoległej");

WatekSymbol[] watekArray = new WatekSymbol[94];
for(int i = 0; i < 94; i++) {
    watekArray[i] = new WatekSymbol(obraz_1, (char)(i + 33));
    watekArray[i].start();
}

for (WatekSymbol watek : watekArray) {
    try {
       watek.join();
    } catch (InterruptedException e) {
       e.printStackTrace();
    }
}
```

e) Dodanie funkcji w klasie ‘Obraz’, która pozwala każdemu wątkowi na wydrukowanie jego fragmentu histogramu w postaci:

```
Wątek 1: & ===============
Wątek 2: % ======
```

```
public synchronized void print_histogram_parallel(char symbol) {
    int count = hist_parallel[symbol - 33];
    if (count != 0) {
       System.out.print("Wątek " + symbol + ": " + symbol + " ");
       for (int i = 0; i < count; i++) {
          System.out.print("=");
       }
       System.out.println();
```

```
    }
}
```

f) Stworzenie metody w klasie ‘Obraz’, która porównuje histogramy równoległy i sekwencyjny

```
public void compare_histograms() {
    for(int i = 0; i < 94; i++) {
       if(histogram[i] != hist_parallel[i]) {
          System.out.println("Niepoprawność dla symbolu: " + tab_symb[i]);
          return;
       }
    }
    System.out.println("Histogramy są zgodne.");
}
```

- Testy poprawności Wariantu 1:
  a) Uruchomienie kodu.
  b) Zweryfikowanie, czy dane z ‘hist_parallel’ są zgodne z danymi z `histogram`.
  c) Zweryfikowanie wydruku graficznego histogramu.

```
4
4
0 w [ ;
o } 0 &
[ P 8 R
X \ e z
```

```
& 1
0 2
8 1
; 1
P 1
R 1
X 1
[ 2
\ 1
e 1
o 1
w 1
z 1
} 1
```

```
Uruchamianie wersji równoległej
Wątek &: & =
Wątek 0: 0 ==
Wątek ;: ; =
Wątek 8: 8 =
Wątek P: P =
Wątek X: X =
Wątek R: R =
Wątek [: [ ==
Wątek \: \ =
Wątek e: e =
Wątek o: o =
Wątek w: w =
Wątek z: z =
Wątek }: } =
Histogramy są zgodne.
```

- Implementacja Wariantu 2 (dekompozycja blokowa):
  a) Skopiowanie plików do nowego katalogu.
  b) Zaimplementowanie wątków na podstawie `Runnable`:
    - Każdy wątek przetwarza blok znaków ASCII.
    - Każdy wątek aktualizuje odpowiednie fragmenty histogramu.

```
class HistogramWorker implements Runnable {
    private Obraz obraz;
    private int startSymbol;
    private int endSymbol;

    public HistogramWorker(Obraz obraz, int startSymbol, int endSymbol) {
        this.obraz = obraz;
        this.startSymbol = startSymbol;
        this.endSymbol = endSymbol;
    }

    @Override
    public void run() {
        obraz.calculate_histogram_parallel_block(startSymbol, endSymbol);
    }
}
```

```
public void calculate_histogram_parallel_block(int startSymbol, int
endSymbol) {
    for (int i = 0; i < size_n; i++) {
       for (int j = 0; j < size_m; j++) {
          for (int k = startSymbol; k < endSymbol; k++) {
             if (tab[i][j] == tab_symb[k]) {
                hist_parallel[k]++;
             }
          }
       }
    }
}
```

c) Zarządzanie wątkami w `main`:
   - Utwórz wątków, przypisując im zakres znaków ASCII.
   - Uruchomienie wątków i oczekiwanie na zakończenie ich pracy.

```
System.out.println("\nSet number of threads");
int num_threads = scanner.nextInt();
Thread[] threads = new Thread[num_threads];

int symbolsPerThread = 94 / num_threads;
for (int i = 0; i < num_threads; i++) {
    int startSymbol = i * symbolsPerThread;
    int endSymbol = (i == num_threads - 1) ? 94 : startSymbol +
symbolsPerThread;
    threads[i] = new Thread(new HistogramWorker(obraz_1, startSymbol,
endSymbol));
    threads[i].start();
}

for (int i = 0; i < num_threads; i++) {
    try {
       threads[i].join();
    } catch (InterruptedException e) {
       e.printStackTrace();
    }
}

System.out.println("\nParallel Histogram:");
obraz_1.print_histogram_parallel();
```

- Funkcja drukująca dla Wariantu 2:

a) Dodanie metody w klasie `Obraz`, która pozwala wątkom na wydruk ich fragmentów histogramu w postaci:

```
Wątek 1: & ==============
Wątek 2: % ======
Wątek 3: @ =========
```

```

```
public void print_histogram_parallel() {
    for (int i = 0; i < 94; i++) {
       if (hist_parallel[i] != 0) {
          System.out.print(tab_symb[i] + " ");
          for (int j = 0; j < hist_parallel[i]; j++) {
             System.out.print("=");
          }
          System.out.print("\n");
       }
    }
}
```

- Testy poprawności Wariantu 2:
  a) Uruchomienie kodu dla różnych zakresów ASCII.
  b) Porównanie tablicy `hist_parallel` z `histogram` w celu sprawdzenia poprawności wyników.
  c) Zweryfikowanie czy wydruki wątków są poprawne i nie zakłócają się nawzajem.

```
4
4
d M < 3
& o , F
u v j 7
z [ = h
```

```
& =
, =
3 =
7 =
< =
= =
F =
M =
[ =
d =
h =
j =
o =
u =
v =
z =
```
```

3. Wnioski

- Działanie programu sekwencyjnego:
  a) Kod sekwencyjny poprawnie oblicza histogram dla małych i dużych tablic.
  b) Liczba wystąpień każdego znaku w tablicy ‘tab’ zgadza się z wartościami w tablicy ‘histogram’.
  c) Działanie sekwencyjne jest proste do zaimplementowania, ale nieefektywne przy dużych tablicach.

- Działanie programu równoległego:

a) Wariant równoległy znacząco przyspiesza obliczanie histogramu dla dużych tablic, szczególnie przy większej liczbie znaków ASCII.
b) Poprawność wyników została zweryfikowana przez porównanie tablic ‘histogram’ (sekwencyjnego) i `hist_parallel` (równoległego). Obie tablice zwracają identyczne wyniki.

- Synchronizacja wątków:
  a) Synchronizacja dostępu do danych była konieczna, aby uniknąć konfliktów podczas aktualizacji tablicy ‘hist_parallel’.
  b) Synchronizowane metody klasy ‘Obraz’ pozwoliły na prawidłowe obliczanie fragmentów histogramu przez wiele wątków jednocześnie.

- Porównanie wariantów równoległych:
  a) Wariant 1 (każdy wątek odpowiada za jeden znak):
     - Łatwiejszy w implementacji, ponieważ każdy wątek obsługuje jedną literę.
     - Wydajność może być ograniczona przy dużej liczbie znaków ASCII (liczba wątków równa liczbie znaków).
     - Efektywny dla tablic zawierających mało różnorodne znaki.
  b) Wariant 2 (dekompozycja blokowa):
     - Bardziej elastyczny – mniejsza liczba wątków, lepsze wykorzystanie zasobów systemowych.
     - Równomierny podział obciążenia między wątki.
     - Wymaga bardziej złożonego zarządzania zakresem znaków w poszczególnych wątkach.

- Wydajność:
  a) Kod równoległy działa szybciej dla tablic o dużych rozmiarach (np. 1000x1000).
  b) Przy małych tablicach (np. 10x10) narzut związany z tworzeniem wątków sprawia, że kod równoległy jest wolniejszy od sekwencyjnego.

Laboratorium nr. 7

1. Cel:

   - nabycie umiejętności pisania programów w języku Java z wykorzystaniem puli wątków

2. Wykonanie i wyniki

   - Utworzenie katalogu roboczego
   - Napisanie sekwencyjnego programu do obliczania całki metodą trapezów z użyciem klasy ‘Calka_callable’

```
public class LiczenieCalkiSekwencyjne {
    public static void main(String[] args) {
       double start = 0;
       double end= Math.PI;
       double dx = 0.000001;
       Calka_callable calka = new Calka_callable(start, end, dx);
       System.out.println("Wynik calki sekwencyjnie: " +
calka.compute_integral());
    }
}
```

   - Uruchomienie programu

```
Creating an instance of Calka_callable
xp = 0.0, xk = 3.141592653589793, N = 3141593
dx requested = 0,000001, dx final = 0,0000009999998897
Wynik calki sekwencyjnie: 1.9999999999997575

Process finished with exit code 0
```

- Pobranie i uruchomienie paczki testowej
- Modyfikacja programu na wersję równoległą z wykorzystaniem ExecutorService:
    a) Podzielenie przedziału całkowania (0, Math.PI) na tyle podprzedziałów, ile jest zadań. Szerokość podprzedziałów jest niezależna od parametru ‘dx’.
    b) Tworzenie zadań i modyfikacja klasy ‘Calka_callable’
        - Utworzenie obiekt klasy ‘Calka_callable’ dla każdego podprzedziału i przekazanie go do puli wątków.
        - Zmodyfikowanie klasy ‘Calka_callable’:
        - Odkomentowanie nagłówka umożliwiającego wykorzystanie obiektu jako zadania w puli wątków.
        - Uzupełnienie kodu klasy o wymaganą przez interfejs funkcję ‘call()’.

    c) Niezależność parametrów
        - Liczba wątków i liczba zadań są niezależne.
        - Liczba wątków powinna być związana z liczbą rdzeni procesora.
        - Liczba zadań powinna być kilka razy większa od liczby wątków, aby równoważyć obciążenie.

    d) Tworzenie i wykonywanie zadań
        - Zadania są tworzone i przekazywane do wykonania w jednej pętli.

    e) Odbieranie wyników
        - Korzystając z interfejsu ’Future’, wyniki są odbierane w oddzielnej pętli, aby umożliwić działanie równoległe.

- Kod programu
    a) Klasa ‘Liczeniecalki’

```
import java.util.concurrent.*;
import java.util.List;
import java.util.ArrayList;

public class Liczeniecalki {

    private static final int NTHREADS = 10;
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(NTHREADS);
//tworzenie wykonawcy o zadanej puli wątków
        List<Future<Double>> listaWynikow = new
ArrayList<Future<Double>>(); // lista w ktorej zapisujemy obiekty future

    //Przedział
    double start = 0;
    double end = Math.PI;

    double task = 50; //ilość zadań
    double dx = 0.000001; //dokładność

    double przedzialPerZadanie = (end - start) / task; //przedział dla
każdego pojedynczego zadania

    double kX = przedzialPerZadanie; //zmienna pomocnicza

    for (double pX = start; pX < end; pX += przedzialPerZadanie){

       if(kX > end){kX = end;}  // w przypadku bledu zaokraglen ten if to
wychwyci

       Callable<Double> calkacallable = new Calka_callable(pX, kX, dx);
       Future<Double> future = executor.submit(calkacallable);
       listaWynikow.add(future);
       kX += przedzialPerZadanie; //przejscie na inny przedział dla
kolejnego wątku z puli
    }

    double wynik = 0;

    for (Future<Double> future : listaWynikow){
       try {
       wynik += future.get(); // uzyskanie wyników metody call
       }
       catch (InterruptedException | ExecutionException e){
          e.printStackTrace();
       }

    }
       executor.shutdown();

    // Wait until all threads finish
    while (!executor.isTerminated()) {}
    System.out.println("Finished all threads");
    System.out.println("Wynik calkowania rownolegle: " + wynik);

    }
   }
```

b) Kod dodany do klasy ‘Calka_callable’

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

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

- Uruchomienie i sprawdzenie wyników z metodą sekwencyjną
  a) Uruchomienie zmodyfikowanego programu dla 10 zadań.

```
Creating an instance of Calka_callable
xp = 0.0, xk = 0.3141592653589793, N = 314160
dx requested = 0,000001, dx final = 0,0000009999976616
Creating an instance of Calka_callable
xp = 0.3141592653589793, xk = 0.6283185307179586, N = 314160
dx requested = 0,000001, dx final = 0,0000009999976616
Creating an instance of Calka_callable
xp = 0.6283185307179586, xk = 0.9424777960769379, N = 314160
dx requested = 0,000001, dx final = 0,0000009999976616
Creating an instance of Calka_callable
xp = 0.9424777960769379, xk = 1.2566370614359172, N = 314160
dx requested = 0,000001, dx final = 0,0000009999976616
Creating an instance of Calka_callable
xp = 1.2566370614359172, xk = 1.5707963267948966, N = 314160
dx requested = 0,000001, dx final = 0,0000009999976616
Creating an instance of Calka_callable
xp = 1.5707963267948966, xk = 1.8849555921538759, N = 314160
dx requested = 0,000001, dx final = 0,0000009999976616
Creating an instance of Calka_callable
xp = 1.8849555921538759, xk = 2.199114857512855, N = 314160
dx requested = 0,000001, dx final = 0,0000009999976616
Creating an instance of Calka_callable
xp = 2.199114857512855, xk = 2.5132741228718345, N = 314160
dx requested = 0,000001, dx final = 0,0000009999976616
Creating an instance of Calka_callable
xp = 2.5132741228718345, xk = 2.827433388230814, N = 314160
dx requested = 0,000001, dx final = 0,0000009999976616
```

```
xp = 2.827433388230814, xk = 3.141592653589793, N = 314160
dx requested = 0,000001, dx final = 0,0000009999976616
Finished all threads
Wynik calkowania rownolegle: 1.9999999999998113

Process finished with exit code 0
```

3. Wnioski

- Poprawność obliczeń
  a) Wersja sekwencyjna programu dostarczyła wyniki zgodne z oczekiwaniami, co zostało zweryfikowane przez porównanie z dokładnym rozwiązaniem analitycznym dla funkcji testowej `sin(x)` w przedziale (0, pi).
  b) Wersja równoległa programu również zwróciła poprawne wyniki, zgodne z wersją sekwencyjną, co świadczy o prawidłowym podziale pracy na podprzedziały i ich sumowaniu.

- Wydajność
  a) Zastosowanie puli wątków znacząco przyspieszyło obliczenia dla dużej liczby zadań w porównaniu z wersją sekwencyjną, szczególnie na maszynach z wielordzeniowymi procesorami.
  b) Wydajność skalowała się wraz ze wzrostem liczby wątków, ale przy nadmiernie dużej liczbie wątków względem dostępnych rdzeni zauważono efekt zmniejszenia efektywności.

- Balans obciążenia

  - Dzięki podziałowi przedziału całkowania na wiele zadań obciążenie procesora zostało dobrze zbalansowane.

  - Nawet jeśli niektóre zadania wymagały minimalnie większej liczby operacji, mechanizm kolejkowania zadań w ‘ExecutorService’ umożliwił efektywne wykorzystanie zasobów.