Patryk Fiałkowski, Informatyka techniczna niestacjonarnie

Programowanie Równoległe, gr.1

Sprawozdanie z laboratorium 6 i 7 (trzecie zajęcia)

Celem dzisiejszych zajęć była nauka programowania równoległego w Javie, rozumiejąc metodykę działania, której nauczyliśmy w języku C przekładamy już zdobytą wiedzę na nowe narzędzia i język.

Lab6

W ramach lab6 mieliśmy program, który generuje tablicę pseudolosowych znaków ASCII o wielkości zadanej przez użytkownika. W gotowej metodzie mieliśmy sekwencyjne wyliczenie występowania wszystkich 94 znaków w tablicy. Ten gotowy histogram z wyliczeń sekwencyjnych przez resztę działań będzie nam służył do weryfikacji działania naszych implementacji zrównoleglenia zadania.

W moim przypadku wykonane zostały 3 metody:

- 1 wątek wyszukuje 1 wyznaczony znak w tablicy

- 1 wątek wyszukuje wyznaczoną ilość znaków w tablicy

- 1 wątek szuka wszystkich znaków w pojedynczym wierszu tablicy znaków pseudolosowych

Ostatnia metoda została przeze mnie dodana w ramach pracy domowej.

Jako, że każda z tych implementacji wstawiała swoje wyliczenia do tego samego histogramu równoległego (tablica parralelHistogram w klasie Obraz) przed każdym testem tablica była zerowana i po każdym wyliczeniu wypisywania i porównywana. W pierwszym wypadku stworzyliśmy klasę Watek, która dziedziczy z Thread, w prostym tłumaczeniu posiada te same metody i atrybuty co wątek w Javie. Przekazujemy obiektowi klasy Watek obiekt klasy Obraz oraz konkretny indeks szukanego znaku w metodzie run() uruchamia on nowe metody z klasy Obraz, które wyliczają histogram dla wskazanego znaku i wypisują histogram dla wskazanego znaku. Warto zauważyć, że o ile wątki nie będą „przeszkadzać sobie" przy wyliczaniu występowania danego znaku, bo każdy wątek ma swój znak, który już żaden inny nie dostanie, o tyle metoda wypisywania histogramu dla danego znaku wymaga synchronizacji, gdyż w danym momencie tylko jeden wątek może wypisywać na ekran.

Porównanie naszych histogramów wykonujemy dopiero po zakończeniu pracy wątków co sprawdzamy osobą pętlą, dzieje się tak dla wszystkich 3 sposobów.

W drugim przypadku wykorzystać możemy te same metody, ale tworzymy tym razem klasę, która implementuje interfejs Runnable, dzięki czemu posiada ona metodę run(). Tym samym tworząc nowy wątek przekazujemy mu nasz obiekt RunnableClass z jego argumentami i uruchamiamy (start()). Tym razem użytkownik podaje ilość wątków, która będzie tworzyć histogram, tym samym nasza nowa klasa otrzyma przedział tablicy znaków, które ma wyszukać. W naszym przypadku dzieli tablicę znaków na równe bloki, z wyłączenie ostatniego bloku, który może być większy. W metodzie run() for wykonujemy działania identyczne dla poprzedniego zadania przez cały wskazany przedział. Warto zwrócić uwagę na wyświetlanie tablicy, które pokazuje nam jak nieregularność w jakiej wątki kończą swoją pracę.

W trzecim przypadku musimy zwrócić uwagę na fakt, że wiele wątków może chcieć zwiększyć ilość danego znaku w histogramie w tym samym czasie, dlatego tworzymy nową synchronizowaną metodę, która wylicza histogram dla danego wiersza, takie rozwiązanie spowalnia sam proces, ale zapewnia

bezpieczeństwo danych. W tym rozwiązaniu wyświetlamy częściowy histogram wyliczony przez każdy wątek, który jest dodawany do głównego oraz całościowy histogram po zakończonej pracy wszystkich wątków.

Lab7

W lab7 dalej tworzyliśmy w języku Java, ale z użyciem executors. W przykładowym programie tworzymy nasz ThreadPhool, który przypisany jest do stworzonego executora, któremu następnie wrzucaliśmy konkretne działania, w tym wypadku proste counter (każde wywołanie po prostu inkrementuje dany counter). Warto zwrócić uwagę na to, że nasz executor musi zostać wyłączony, inaczej nasz program może działa w nieskończoność, po jego zamknięciu wypisujemy nasze countery.

W naszym programie podobnie jak na wcześniejszych ćwiczeniach, tylko tym razie w Javie z użycemu ExecutorService wyliczamy całkę. Każdy nasz wątek oblicza całkę częściową. A po zakończeniu ich pracy dodajemy te częściowe całki do całki końcowej (sum_calka), oczywiście pamiętamy o zakończeniu pracy naszego executora.

Wnioski:

Język Java pozwala na programowanie równoległe różną metodyką i dostępnymi bibliotekami, co pozwala nam dostosować nasz program pod dane potrzeby.

Tak samo jak w C, tak w Javie musimy wiedzieć, gdzie w naszym programie są sekcje krytyczne, które musimy w odpowiedni sposób zabezpieczyć

W języku obiektowym programowanie równoległe również jest w pełni możliwe i wygodne w użyciu.

Sposób zrównoleglenia naszego programu powinien być dobrany do potrzeb danego programu/projektu.

W każdym momencie w języku Java pracując na wątkach możemy sprawdzić w jaki wątku się znajdujemy metodą currentThread().getName().

Załączniki:

W związku z długością zwracanych histogramów w sprawozdaniu będą jedynie ich fragmenty po wypisaniu przez program, dodatkowo dodaję linki do obu projektów na platformie GitHub:

Lab6 - https://github.com/PFialkowskiAGH/ParralelProgramingJava

Lab7 - https://github.com/PFialkowskiAGH/ExecutorJava

Podanie wielkości tablicy i wypisanie jej:

```
Set image size: n (#rows), m(#kolumns)
5
5
= @ 1 F =
g 5 t _ 1
S " T x z
} ~ i h v
^ [ C o g
```

Wypisanie histogramu sekwencyjnego

```
! 0
" 1
# 0
$ 0
% 0
& 0
```

Wypisanie histogramu zrównoleglonego, każdy wątek zajmuje się jednym znakiem (każdy znak równa się to jedno wystąpienie)

```
----------------
Thread-0: !
Thread-12: -
Thread-8: )
Thread-1: "=
Thread-10: +
Thread-9: *
Thread-11: ,
Thread-7: (
```

Weryfikacja Histogramu po zakończeniu jego wypisywania

```
Thread-81: r
Thread-80: q
Histogramy są takie same
----------------------------
```

Podanie ilości wątków i wypisanie histogramu

```
Set number of threads
4
Thread-94: !
Thread-94: "=
Thread-94: #
Thread-94: $
Thread-95: 8
Thread-95: 9
```

Kod klasy Obrazek (dodane metody)

```
2 usages  Ayuzawa
public void clear_only_parallelHistogram() { for(int i=0;i<94;i++) parallelHistogram[i]=0; }
```

```
2 usages  Patryk
public void calculate_histogram_for_char(int searchedChar)
{
    for(int i=0;i<size_n;i++) {
        for(int j=0;j<size_m;j++) {
            if(tab[i][j] == tab_symb[searchedChar]) parallelHistogram[searchedChar]++;
        }
    }
}
2 usages  Patryk
public synchronized void print_histogram_for_char(int searchedChar)
{
    System.out.print(Watek.currentThread().getName() + ": ");
    System.out.print(tab_symb[searchedChar]);
    for (int i = 0; i < parallelHistogram[searchedChar]; i++)
    {...}
    System.out.print("\n");
}
```

```
1 usage  Ayuzawa
public synchronized void calculate_print_histogram_for_one_row(int searchedRow)
{
    int[] partParrarelHistogram = new int[94];
    for(int i=0;i<94;i++) partParrarelHistogram[i]=0;

    for(int j=0;j<size_m;j++) {
        for(int k=0;k<94;k++) {
            if(tab[searchedRow][j] == tab_symb[k]) partParrarelHistogram[k]++;
        }
    }

    for (int i=0;i<94;i++)
    {
        System.out.print(tab_symb[i]+" "+partParrarelHistogram[i]+"\n");
        parallelHistogram[i] += partParrarelHistogram[i];
    }
}
```

```
1 usage  Ayuzawa
public synchronized void calculate_print_histogram_for_one_row(int searchedRow)
{
    int[] partParrarelHistogram = new int[94];
    for(int i=0;i<94;i++) partParrarelHistogram[i]=0;

    for(int j=0;j<size_m;j++) {
        for(int k=0;k<94;k++) {
            if(tab[searchedRow][j] == tab_symb[k]) partParrarelHistogram[k]++;
        }
    }

    for (int i=0;i<94;i++)
    {
        System.out.print(tab_symb[i]+" "+partParrarelHistogram[i]+"\n");
        parallelHistogram[i] += partParrarelHistogram[i];
    }
}
```

```
3 usages  Patryk
public void compareHistogram()
{
    boolean isSame = true;
    for(int i=0;i<94;i++)
    {
        if (histogram[i] != parallelHistogram[i]) isSame = false;
    }
    if (isSame) System.out.println("Histogramy są takie same");
    else System.out.println("Histogramy nie są takie same");
}
```

Klasa Watek

```
public class Watek extends Thread
{
    3 usages
    Obraz obraz;
    3 usages
    int searchedChar;
    1 usage  Patryk
    public Watek(int i, Obraz obraz)
    {
        this.obraz = obraz;
        this.searchedChar = i;
    }
    Patryk +1
    public void run()
    {
        obraz.calculate_histogram_for_char(searchedChar);
        obraz.print_histogram_for_char(searchedChar);
    }
}
```

Klasa RunnableClass

```java
public class RunnableClass implements Runnable
{
    3 usages
    Obraz obraz;
    2 usages
    int startChar;
    2 usages
    int endChar;

    2 usages  Ayuzawa +1
    public RunnableClass(int startChar, int endChar, Obraz obraz)
    {
        this.obraz = obraz;
        this.startChar = startChar;
        this.endChar = endChar;
    }

    Patryk +1
    @Override
    public void run()
    {
        for (int i = startChar; i < endChar; i++)
        {
            obraz.calculate_histogram_for_char(i);
            obraz.print_histogram_for_char(i);
        }
    }
}
```

Klasa RunnableClass2

```java
public class RunnableClass2 implements Runnable
{
    2 usages
    Obraz obraz;
    2 usages
    int row;
    1 usage  Ayuzawa
    public RunnableClass2(int row, Obraz obraz)
    {
        this.obraz = obraz;
        this.row = row;
    }
    Ayuzawa
    @Override
    public void run()
    {
        obraz.calculate_print_histogram_for_one_row(row);
    }
}
```

Main – metoda1

```
int num_threads = 94;//scanner.nextInt();

Watek[] newThr = new Watek[num_threads];

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

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

obraz_1.compareHistogram();
```

Main – metoda2

```
System.out.println("Set number of threads");
obraz_1.clear_only_parallelHistogram();
num_threads = scanner.nextInt();
int d = 94 % num_threads;
int r = 94 / num_threads;

Thread[] newThr2 = new Thread[num_threads];

for (int i = 0; i < num_threads; i ++)
{
    if (i == num_threads-1) (newThr2[i] = new Thread(new RunnableClass( startChar: i*r, endChar: i*r+r+d,obraz_1))).start();
    else (newThr2[i] = new Thread(new RunnableClass( startChar: i*r, endChar: i*r+r,obraz_1))).start();
}
for (int i = 0; i < num_threads; i++) {
    try {
        newThr2[i].join();
    } catch (InterruptedException e) {}
}

obraz_1.compareHistogram();
```

Main-metoda3

```
System.out.println("Every row of random charsTab is new Thread");
obraz_1.clear_only_parallelHistogram();

num_threads = n;
Thread[] newThr3 = new Thread[num_threads];

for (int i = 0; i < num_threads; i++)
{
    (newThr3[i] = new Thread(new RunnableClass2(i,obraz_1))).start();
}
for (int i = 0; i < num_threads; i++) {
    try {
        newThr3[i].join();
    } catch (InterruptedException e) {}
}

obraz_1.print_parallelHistogram();
obraz_1.compareHistogram();
```

Przykład Executor

```
Finished all threads

Counter_1: 1000, Counter_2: 1000
```

Wyliczenie całki

```
Creating an instance of Calka_callable
xp = 0.0, xk = 0.3141592653589793, N = 32
dx requested = 0.01, dx final = 0.009817477042468103
Creating an instance of Calka_callable
xp = 0.3141592653589793, xk = 0.6283185307179586, N = 32
```

```
Calka czastkowa: 0.27876601887386937
Calka czastkowa: 0.27876601887386937
Calka czastkowa: 0.14203838107048072
Final calka = 1.999983936164949
```

Kod wyliczania całki

```
private static final int NTHREADS = 10;
3 usages
static List<FutureTask<Double>> tasksTab;

no usages  Ayuzawa
public static void main(String[] args) {
    tasksTab = new ArrayList<FutureTask<Double>>();
    double sum_calka = 0.0;
    double r = Math.PI / NTHREADS;
    ExecutorService executor = Executors.newFixedThreadPool(NTHREADS);

    for (int i = 0; i < NTHREADS; i++)
    {
        Calka_callable clb = new Calka_callable( xp: i*r,  xk: (i+1)*r,  dx: 0.1 / NTHREADS);
        FutureTask ft =new FutureTask<Double>(clb);
        tasksTab.add(ft);
        executor.execute(ft);
    }
```

```
for (FutureTask<Double> futureTask : tasksTab) {
    try
    {
        sum_calka += futureTask.get();
    }
    catch (InterruptedException | ExecutionException e)
    {
        e.printStackTrace();
    }
}
executor.shutdown();
System.out.println("Final calka = " + sum_calka);
```