Grupa lab.
5

Data ćwiczenia
6.11.2023

Laboratorium 6

Imie i nazwisko
Łukasz Sawina

Cel

W tych ćwiczeniach zamiast klasycznego programowania w C zostanie wykorzystany język Java oraz jego mechaniki do pracy z wątkami. Zadanie polega na policzeniu wystąpień znaków obrazie (tablicy n x m znaków) oraz wyświetlenie ich w pewnego rodzaju histogramu.

Wersja sekwencyjna działa na zasadzie trzech pętli, pierwsza przechodzi po wierszach, drugia po kolumnach, trzecia przez tablicę wszystkich znaków. Jak można się domyślić takie działanie jest bardzo obciążające, ponieważ jest to pętla w pętli w pętli, dlatego naszym zadaniem jest zrobienie tego lepiej, przy pomocy zrównoleglenia programu.

W celu porównania wyników oraz upewnienia się, czy nasz program działa poprawnie wyniki histogramu wersji równoległej zapisywane są w nowej tablicy *hist_parallel.* Dodatkowo do klasy Obraz dodałem metodę sprawdzającą zgodność obu histogramów i w przypadku różnicy informującą w którym miejscu jest różnica oraz jaka.

```
public void sprawdz_wynik()
{
   boolean czy_poprawne = true;
   for(int i = 0; i < 94; i++)
      if(hist_parallel[i] != histogram[i])
      {
         czy_poprawne = false;
         System.out.println("Różnica przy znaku " + tab_symb[i] + " dla sekw: " +
histogram[i] + " dla rów: " + hist_parallel[i]);
      }

   if(czy_poprawne)
      System.out.println("Oba histogramy są identyczne");
}
```

Powyższa metoda wywołana jest zawsze na końcu programu, aby upewnić się czy wszystko działa poprawnie.

Dodatkowo do klasy Obraz dopisana została nowa metoda *calculate_histogram_parallel*, która przyjmuje jako parametry informacje o początkach, końcach oraz długości skoków dla wszystkich trzech pętli. Ta wersja metody jest podstawową, niezabezpieczoną, w dalszej części pojawi się nowa zabezpieczająca nasze dane przez ewentualnymi problemami.

```
public void calculate_histogram_parallel(int start_wiersz, int end_wiersz, int
skok_wiersz,
                    int start_kol, int end_kol, int skok_kol,
                    int start_znak, int end_znak, int skok_znak){
  for(int i=start_wiersz;i<end_wiersz;i+=skok_wiersz) {
    for(int j=start_kol;j<end_kol;j+=skok_kol) {
      for(int k=start_znak;k<end_znak;k+=skok_znak) {
        if (tab[i][j] == tab_symb[k]) hist_parallel[k]++;
      }
    }
  }
}
```

Wariant 1 – Każdy wątek otrzymuje jeden znak do sprawdzenia (wykorzystanie dziedziczenia po Thread)

W tej wersji tworzymy dodatkową klasę *Watek*, która dziedziczy po klasie *Thread*. Jej zadaniem jest w metodzie *run()* wywołanie metody obliczania histogramu dla obiektu obrazu. Jednak każdy o obiekt klasy *Watek* otrzymuje swój własny numer ID, który określa jaki znak ma sprawdzać w obrazie.

```
public Watek(int id, Obraz n_obraz)
{
   ID = id;
   obraz = n_obraz;
}

public void run()
{
   obraz.calculate_histogram_parallel(
         0, obraz.getSizeN(), 1,
         0, obraz.getSizeM(), 1,
         ID, ID+1, 1);

   synchronized (obraz){
      obraz.better_print_histogram(ID, ID);
   }
}
```

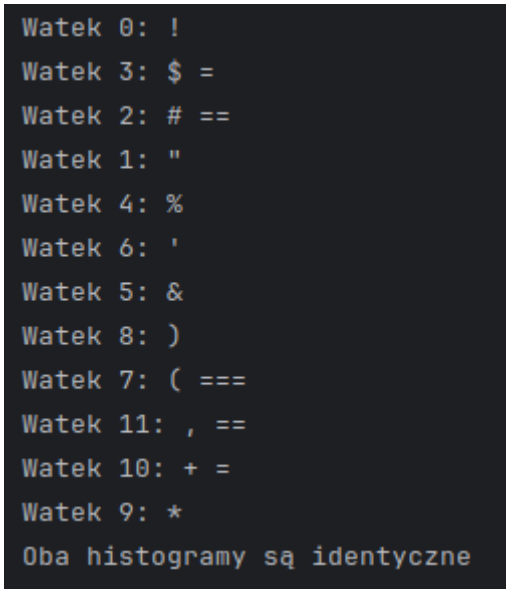
Klasa Watek ma konstruktor, do którego przekazuje się obiekt obrazu na którym ma wykonać histogram oraz jego numer ID. Następnie w metodzie *run* wykonywane są dwie rzeczy, metoda *calculate_histogram_parallel*, w której jak widać pętle po wierszach oraz po kolumnach muszą przejść przez wszystkie elementy. Dopiero pętla przechodząca po znakach ma ustawiony początek na numerze ID wątku oraz przeskakuje o 1 kończąc na kolejnym symbolu. W efekcie pętla przechodząca po znakach wykona się tylko raz dla kazdego elementu obrazu.

Na koniec każdy z wątków wyświetla informacje o ilości wystąpień danego znaku, w metodzie better_print_histogram() przesyłana jest informacja o numerze wątku oraz o numerze symbolu, który zliczał. Cała operacja jest zabezpieczona przy pomocy synchornized(obraz) w celu zabezpieczenia wyświetlania, aby inne wątki nie zepsuły efektu wizualnego.

W efekcie otrzymujemy:

Obraz na jakim sprawdzamy program (w tym przypadku zmiejszyłem ilość możliwych znaków do 12 różnych w celu łatwiejszej prezentacji wyniku), w tym przypadku o wymiarach 3 x 3.

Efekt wyświetlenia histogramu:



Jak widać ilość wystąpień znaków ‘=’ zgadza się z naszym obrazem, dodatkowo informuje nas o tym metoda sprawdzająca poprawność wyników na samym dole.

Nasze obiekty wątków są wywoływane następujaco:

```
System.out.println("Set number of threads");
int num_threads = 12;

 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) {}
 }
```

Ważnym elementem tutaj, jest fakt stałej ilości wątków, nie możemy ich określić sami, ile ma być wykorzystanych, jest to narzucone przez ilość możliwych znaków w obrazie.

Każdy wątek jest tworzony i wywoływana jest na nich od razu metoda *start()*, która następnie wywoła metodę run() automatycznie. Następnie program oczekuje na zkończenie wyrzystkich wątków.

Wariant 2 – Podział blokowy dla znaków (wykorzystanie interfejsu Runnable)

Tym razem program zadziała podobnie, tylko zamiast jednego wątku na każdy znak, podzielimy nasze znaki blokow, w efekcie jeden wątek otrzyma pewien blok znaków do sprawdzenia.

Kolejną różnicą jest wykorzystanie interfejsu Runnable, zamiast dziedziczenia po klasie Thread. W tym przypadku klasa WatekRunnable implementuje nasz interfejs co wymusza na nas utworzenie metody run(), która będzie wykonywałą się równolegle z innymi.

```
public WatekRunnable(int id, int n_start, int n_end, Obraz n_obraz)
{
   obraz = n_obraz;
   ID = id;
   start_i = n_start;
   end_i = n_end;
}
public void run()
{
   obraz.calculate_histogram_parallel(
obraz.getSizeN(), 1,
obraz.getSizeM(), 1,
         start_i, end_i, 1);

   synchronized (obraz)
   {
      for(int i = start_i; i < end_i; i++)
         obraz.better_print_histogram(ID, i);
   }
}
```

W tym przypadku zmienia się nam lekko konstruktor, przyjmuje on teraz dodatkowe dwie informacje, poczatek oraz koniec zakresu do sprawdzania w symbolach. Samo działanie metody *run()* nie zmienia się tak drastycznie, jedyne zmiany to przy przesyłaniu parametrów do pętli przechodzącej po znakach, przekazujemy mu nasz zakres do sprawdzenia oraz przy wyświetlaniu histogramu musimy przesłać wszystkie znaki jakie sprawdzał nasz obiekt.

Różnica również pojawia się przy tworzeniu naszych obiektów WatekRunnable

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

Thread[] NewRunn = new Thread[num_threads];

int offset = (int)ceil(12.0/(double)num_threads);
for (int i = 0; i < num_threads; i++) {
   int start = i*offset;
   int end = (i+1)*offset;
   if(end > 12)
      end = 12;
   (NewRunn[i] = new Thread(new WatekRunnable(i,start, end, obraz_1))).start();
}

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

Tym razem mozemy podać ilość wątków jaką chcemy wykorzystać, następnie tworzona jest tablica obiektów *Thread* i to jest ważne, nie tworzymy tablicy obiektów *WatekRunnable* tylko *Thread*, ponieważ następnie pętli tworzącej wątki będziemy do konstruktora Thread przekazywać nowe obiekty naszej klasy. W taki sposób obiekty *Thread* uruchomią automatycznie metody *run()* z naszych obiektów. Na końcu standardowo oczekiwane jest zakończenie wątków.

Przy tworzeniu każdego wątku dodatkowo obliczane są jego parametry, czyli start oraz koniec. Działa to na klasycznej zasadzie, podziału całego zbioru na ilość wątków z zaokrąglaniem w górę oraz sprawdzenie czy zakres nie został przekroczony przez ostatni wątek.

W efekcie otrzymujemy:

Obraz na jakim zostanie wykonany histogram:

Histogram otrzymany przez podział blokowy dla znaków:

```
Set number of threads
5
Watek 0: !
Watek 0: " =
Watek 0: # =
Watek 3: * =
Watek 3: + =
Watek 3: , ==
Watek 2: '
Watek 2: (
Watek 2: ) =
Watek 1: $ =
Watek 1: % =
Watek 1: &
Oba histogramy są identyczne
```

Jak widać podział został wykonany prawidłowo, dla 5 wątków, otrzymały po 3 elementy, poza ostatnim, ponieważ dla niego nie było już nic do zrobienia. (Niefortunne dobranie ilości znaków nie pozwala mi fajnie pokazać podziału blokowego).

Wariant 3 – Podział cykliczny wierszowy po tablicy (wykorzystanie interfejsu Runnable)

W kolejnych wariantach będziemy wykonywać podział nie ze względu na znaki, a na obraz. Do tego potrzebujemy stworzyć nową wersję funkcji zliczającej znaki, takiej która zrobi to bezpieczenie, ponieważ wszystkie wątki będą działały na tej samej tablicy znaków. W tym celu każdy wątek przy zliczaniu znaków wykona to na lokalnej tablicy, a następnie bezpieczenie dopisze swoje wyniki do globalnej tablicy.

```
public void calculate_histogram_parallel_zabezpieczone(int start_wiersz, int
end_wiersz, int skok_wiersz,
                                int start_kol, int end_kol, int skok_kol,
                                int start_znak, int end_znak, int skok_znak){
   int [] lok_hist = new int[12];

   for(int i=start_wiersz;i<end_wiersz;i+=skok_wiersz) {
      for(int j=start_kol;j<end_kol;j+=skok_kol) {
         for(int k=start_znak;k<end_znak;k+=skok_znak) {
            if (tab[i][j] == tab_symb[k]) lok_hist[k]++;
         }
      }
   }

   synchronized(hist_parallel)
   {
      for(int i = 0; i < 12; i++)
         hist_parallel[i] += lok_hist[i];
   }
}
```

Powyższa funkcja jest bezpieczeniejszą wersją poprzedniej zliczającej ilość znaków. Po zliczeniu w tablicy lokalnej, wartości są przenoszone do głównej tablicy, co jest zabezpieczone przez *synchonized()*.

Do obecnego wariantu utworzona została kolejna klasa, która w tym przypadku zrobi podział cykliczny po wierszach, każdy wątek będzie miał za zadanie sprawdzić cały jeden wiersz, a następnie przejść do kolejnego dostępnego.

```
public WatekRunnable1D(int id, int n_l_w, Obraz n_obraz)
{
   obraz = n_obraz;
   ID = id;
   l_w = n_l_w;
}

public void run() {
   obraz.calculate_histogram_parallel_zabezpieczone(
         ID, obraz.getSizeN(), l_w,
         0, obraz.getSizeM(), 1,
         0, 12, 1);
}
```

Tym razem konstruktor przyjmuje ID, obraz oraz informację o tym ile jest wątków. Różnica w działaniu *run()* polega na tym, że wątek zaczyna działanie numeru wiersza, który odpowiada jego ID i przeskakuje o liczbę wierszy dalej, tak aż nie skończy się ilość wierszy.

W efekcie otrzymujemy:

Obraz dla jakiego jest wykonywany histogram.

Wynik jako histogram (tym razem bez informacji o wątku)

```
Set number of threads
2
Znak: ! =
Znak: " =
Znak: #
Znak: $ ===
Znak: %
Znak: &
Znak: ' =
Znak: (
Znak: )
Znak: * ==
Znak: + =
Znak: ,
Oba histogramy są identyczne
```

Jak można zauważyć wszystko się zgadza. Dodatkowa różnica w tym działaniu polega na wyświetleniu histogramu. Jest on wyświetlany cały na końcu wykonania zadania, ponieważ nie byłoby sensu wykonywać to tak jak poptrzednio.

Wyświetlanie znaków działa na zasadzie podwójnej pętli.

```
public void prittier_print_histogram()
{
   for(int i = 0; i < 12; i++)
   {
      char znak = tab_symb[i];
      System.out.print("Znak: " + znak + " ");
      for(int j=0;j<hist_parallel[i];j++) {
         System.out.print('=');
      }
      System.out.println();
   }
}
```

Wariant 4 – Podział blokowy kolumn (wykorzystanie interfejsu Runnable)

Kolejny program jest podobny do wcześniejszego, różnica jedynie pojawia się przy podziale obrazu. Tym razem dzielimy kolumny obrazu na bloki i każdy wątek otrzymuje pewien blok do sprawdzenia.

```
public WatekRunnable1DBlokowy(int id, int n_start, int n_end, Obraz n_obraz)
{
   obraz = n_obraz;
   ID = id;
   start = n_start;
   end = n_end;
}

public void run() {
   obraz.calculate_histogram_parallel_zabezpieczone(
         0, obraz.getSizeN(), 1,
         start, end, 1,
         0, 12, 1);
}
```

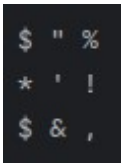
Konstruktor, podobnie jak w podziale blokowym dla znaków, przyjmuje informację o początku oraz końcu bloku na którym ma operować. Te informacje przesyła do funkcji obliczajacej historam.

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

Thread[] NewRunn = new Thread[num_threads];
int offset = (int)ceil(obraz_1.getSizeM()/(double)num_threads);

for (int i = 0; i < num_threads; i++) {
   int start = i*offset;
   int end = (i+1)*offset;
   if(end > obraz_1.getSizeM())
      end = obraz_1.getSizeM();
   (NewRunn[i] = new Thread(new WatekRunnable1DBlokowy(i, start, end,
obraz_1))).start();
}
for (int i = 0; i < num_threads; i++) {
    try {
       NewRunn[i].join();
    } catch (InterruptedException e) {}
 }

obraz_1.prittier_print_histogram();
```

Tutaj również na początku muszą być obliczone wartości początku oraz końca dla bloku. Ponownie jest to podzielenie ilosci kolumn przez ilość wątków z zaokrągleniem w górę oraz sprawdzenie czy nie wychodzimy poza zakres.

W efekcie otrzymujemy:

Obraz, dla którego wykonywany jest histogram



Histogram dla obrazu, jak widać histogram jest zgodny z prawdą.

```
Set number of threads
2
Znak: ! =
Znak: " =
Znak: #
Znak: $ ==
Znak: % =
Znak: & =
Znak: ' =
Znak: (
Znak: )
Znak: * =
Znak: +
Znak: , =
Oba histogramy są identyczne
```

Przez brak wyświetlenia jaki wątek jaką kolumnę wykonuje nie widać dokładnie tego efektu, ale wiemy, że pierwszy wątek wykona obliczenia na kolumnach 1 i 2, a drugi na 3 kolumnie.

Wariant 5 – Podział 2D (wykorzystanie interfejsu Runnable)

W tym wariancie nie będziemy się ograniczać do podziału tylko jednego wymiaru, tutaj cały obraz zostanie podzielony na mniejsze obrazy. Taki podział jest jendak trudniejszy, ponieważ trzeba zadbać o dobre zakresy w dwóch wymiarach.

```
public WatekRunnable2D(Obraz n_obraz, int nw_start, int nw_end, int nk_start, int
nk_end)
{
   obraz = n_obraz;
   w_start = nw_start;
   w_end = nw_end;
   k_start = nk_start;
   k_end = nk_end;
}


@Override
public void run() {
   obraz.calculate_histogram_parallel_zabezpieczone(
         w_start, w_end, 1,
         k_start, k_end, 1,
         0, 12, 1);
}
```

Tym razem konstruktor otrzymuje informacje o początku oraz końcu zakresu w wierszach oraz kolumnach i dla takich przedziałów wywołuje obliczanie histogramu.

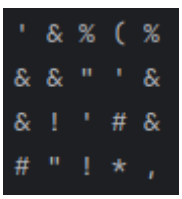
Więcej rzeczy jednak dzieje się w głównej metodzie, gdzie są tworzone wątki.

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

Thread[] NewRunn = new Thread[num_threads];
int w_offset = (int)ceil(obraz_1.getSizeN()/((double)num_threads/2));
int k_offset = (int)ceil(obraz_1.getSizeM()/((double)num_threads/2));

for (int i = 0, k = 0; i < num_threads/2; i++) {
   for(int j = 0; j < num_threads/2; j++, k++)
   {
      int w_start = i*w_offset;
      int w_end = (i+1)*w_offset;
      int k_start = j*k_offset;
      int k_end = (j+1)*k_offset;
      if(w_end > obraz_1.getSizeN())
         w_end = obraz_1.getSizeN();

      if(k_end > obraz_1.getSizeM())
         k_end = obraz_1.getSizeM();

      (NewRunn[k] = new Thread(new WatekRunnable2D(obraz_1, w_start, w_end, k_start,
k_end))).start();
   }
}
for (int i = 0; i < num_threads; i++) {
    try {
       NewRunn[i].join();
    } catch (InterruptedException e) {}
 }

obraz_1.prittier_print_histogram();

obraz_1.sprawdz_wynik();
```

Przede wszystkim poprawnie obliczany musi być offset, w tym przypadku jest on osobno liczony dla obu wymiarów oraz nie z całej ilości wątków, a z ich połowy (w tym programie ważne, aby ilość wątków była parzysta, niestety nie udało mi się rozwiązać tej kwesti). Następnie dla każdego wątku obliczane są początki oraz końce dla wierszy i kolumn. Całość dzieje się w podwójnej pętli, tak aby wszystkie obszary w obrazie zostały obliczone. Na końcu sprawdzane są wartości, czy nie wychodzimy poza obraz w obu kierunkach i tworzone są wątki z przekazanymi parametrami.

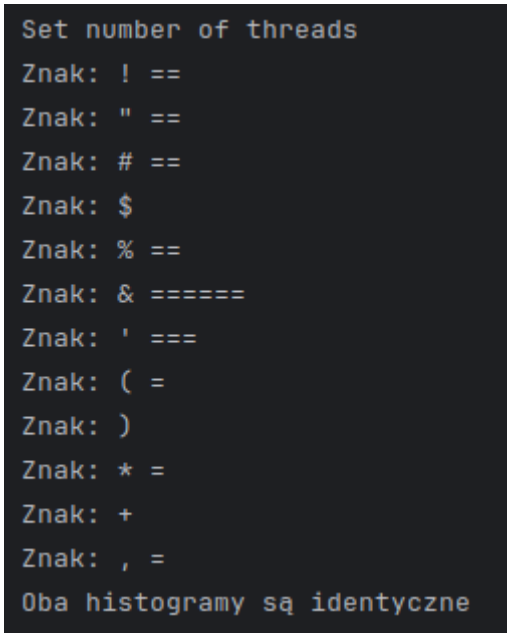
W efekcie otrzymujemy:

Obraz dla którego wykonujemy histogram



Jak widać jest to obraz 4 x 5 a do programu wykorzystane zostały 4 wątki, daltego przy źle określonych obszarach program może źle działać.

Histogram dla obrazu (obliczony na 4 wątkach)



Tak jednak nie jest przez odpowiednie dobranie zakresów dla wątków. Histogram jest zgodny z prawdą.

Wnioski

Jak widac nie tylko w C jest możliwość wykonywania operacji na kilku wątkach, obecnie większość języków programowania pozwala na takie działanie. W naszym przypadku była to Java, w której, aby wykonać operacje równolegle możemy zastosować dwa podejscia.Utworzyć klasę dziedziczącą po *Thread* lub implementującą interfejs *Runnable.*

Obie klasy muszą posiadać w sobie metodę *run()*, które wywoływane są automatycznie przez metodę *start()*. Jednak tworzenie obiektów jest inne, w przypadku dziedziczenia wystarszy utworzyć nowy obiekt i wywołać na nim metodę *start()*, co automatycznie uruchomi *run()*. W przypadku interfejsu obiekty są tworzone w konstruktorach klas *Thread*, na których są wywoływane metody *start()*.

Jak widać znając różne sposoby na zrównoleglenie problemu oraz zabezpieczenia danych można je wykonywać prawie identycznie w różnych językach, głównymi różnicami jest sama składnia języków. W tym przypadku było to tworzenie i uruchamianie obiektów oraz zabezpieczanie danych.

W C na zabezpieczenie danych musimy wykorzystać *mutexy*, tutaj mamy jednak dwa podejścia, albo tworzymy blok *synchronized*, który zabezpiecza przekazaną mu zmienną, albo określamy metodę jako *synchronized* i jest ona wykonywana synchronicznie.