

Marek, Kubicki

Programowanie równoległe.

Sprawozdanie z laboratorium 6,7,8 - Java.

1. Celem tych zajęć było zapoznanie się z działaniem systemów w javie odpowiedzialnych za zrównoleglanie pracy programu.

Java jest językiem obiektowym, więc narzędzia odpowiedzialne za obsługę wielowątkowości są konstrukcjami obiektowymi. Wątki są obiektami klasy **Thread**, wielowątkowość jest wbudowana w podstawy języka, w klasę **Object** (dziedziczy po niej każda klasa w Javie) za pomocą metod: **wait**, **notify** i **notifyAll**. Wątek może być zatrzymany poprzez wywołanie metody **wait**, a następnie wznowiony za pomocą **notify** lub **notifyAll**. Pierwsza z tych dwóch metod wznowia jeden wątek oczekujący za pomocą **wait** (nie da się określić który), druga wznowia wszystkie wątki.

Wątki w javie mogą być uruchomione na dwa sposoby, oba związane z klasą **Thread**.

1. Poprzez obiekt klasy **Thread**, implementującą interfejs **Runnable** i implementującą metodę **run**, poprzez przekazanie argumentu do metody **start**.

2. Poprzez obiekt klasy dziedziczącej po **Thread**, przeciążając metodę **run**, poprzez przekazanie argumentu do metody **start**.

W przypadku implementacji **Runnable** można dziedziczyć po innych klasach, a w przypadku dziedziczenia po **Thread** nie można.

Klasa **Thread** ma wiele metod umożliwiających poprawne implementowanie programu wielowątkowego, poniżej przedstawiłem te wykorzystywane na zajęciach. Metody **start** i **run** odpowiedzialne za uruchamianie wątku, **start** jest wywoływane przez rodzica, następnie **start** wywołuje **run**, które wykonuje odpowiednie zadania. Metoda **sleep** usypia wątek na czas podany w argumencie (w milisekundach). Metody **join** i **interrupt** są odpowiedzialne za zakończanie pracy wątku. **Join** oczekuje na naturalne zakończenie pracy wątku, a **interrupt** przerwia działanie wątku kiedy ten znajduje się w metodzie dającej się przerwać np. **wait**.

Mechanizmem umożliwiającym bezpieczne obsługiwanie sekcji krytycznych zamek monitorowy, występujący w każdym obiekcie. Rozpoczęcie metody oznaczonej przez **synchronized** (lub bloku synchronizowanego) blokuje inne wątki przed rozpoczęciem innych podobnych metod. Wątki oczekują na taką metodę w kolejce i są wznowiane od pierwszego kiedy obiekt zajmujący zamek go zwolni.

W przypadku zrównoleglenia z użyciem znaczającej liczby wątków java posiada dodatkowe narzędzia w postaci puli wątków. Służą do tego obiekty klasy **ThreadPoolExecutor** za pomocą dostarczonych im obiektów klas implementujących **Runnable** lub **Callable**. To rozwiązanie unika niepotrzebnej dealokacji i alokacji pamięci poprzez używanie tych samych wątków przydzielając im zadania z kolejki. Zarządzaniem wątkami i zadaniami zajmuje się exekutor (**ExecutorService**). Exekutor może mieć stałą liczbę wątków, lub zarządzać ich ilością dynamicznie.

Do przekazywania wyniku i sprawdzenia zakończenia działania służą obiekty **Future**. Za ich pomocą można sprawdzać, czy zadanie zostało zakończone (**isDone**), odbieranie wyniku (**get**) z oczekiwaniem na zakończenie pracy wątku lub przez określony czas (zależy od argumentów), anulowanie zadań (**cancel**) i sprawdzanie, czy zadania zostały anulowane (**isCancelled**)

Fork/Join jest mechanizmem podobnym do powyżej opisanego **ThreaPool**. Polega on na rekursywnym podziale zadań na mniejsze zadania (**fork**) i następny pobieraniu wyników (**join**). Jest to mechanizm pomocny w podziale zadań gdy te są wystarczająco duże (schemat dziel i zwyciężaj).

Condition Interface stanowi sposób na tymczasowe zatrzymanie działania wątku bez użycia sekcji krytycznej (**synchronized**). Działają podobnie do mechanizmu **wait/notify**, ale z dodatkową możliwością utworzenia wielu kolejek. Wątek oczekujący na danym warunku (**await**) oczekuje na zawiadomienie na warunku (**notify**) przed wznowieniem działania.

laboratorium 6

Na laboratorium 6 zadaniem było zmodyfikowanie podanego programu. Program ten zliczał znaki w podanej tablicy sekwencyjnie. Celem było zrównoleglenie działanie tego programu za pomocą 2 wariantów.

Wariant 1: Każdy wątek odpowiedzialny jest za jeden znak

Tworzenie wątków i oczekiwanie na wyniki:

```
System.out.println("Set number of threads char");
int num_threads = scanner.nextInt();
znak_his_row[] NewThr = new znak_his_row[num_threads];

for (int i = 0; i < num_threads; i++) {
    NewThr[i] = new znak_his_row(i, obraz_1);
    NewThr[i].start();
}
for (int i = 0; i < num_threads; i++) {
    try {
        NewThr[i].join();
        if (Thread.interrupted()) {
            throw new InterruptedException();
        }
    } catch (InterruptedException e) {}
}
```

Ciało klasy extendującej **Thread**:

```
public /*static*/ class znak_his_row extends Thread {
    int i;
    Obraz obraz_1;
    public znak_his_row(int ia, Obraz ob)
    {
        i = ia;
        obraz_1 = ob;
    }
    @Override
    public void run()
    {
        obraz_1.calculate_histogram_char(i);
        obraz_1.print_histogram_char(i);
    }
}
```

```
}
```

Każdy wątek zlicza znak podany za pomocą **int** i wywołując odpowiednie metody przekazanego mu obiektu zawierający histogram i tablice do której zapisywane są wyniki.

```
Set number of threads char
```

```
8
```

```
Watek 0: !      #
Watek 7: (      ##
Watek 4: %      #
Watek 6: '      #
Watek 5: &      #
Watek 3: $      #
Watek 2: #      #
Watek 1: "      #
```

Każdy wątek zlicza tylko jeden znak, brak pełnego histogramu w przypadku niewystarczającej liczby wątków.

Wariant 2:

Wariant 1: Każdy wątek odpowiedzialny jest za przedział znaków

Tworzenie wątków i oczekiwanie na wyniki:

```
System.out.println("\nSet number of threads bound");
int num_threads_b = scanner.nextInt();
int krok = 94/num_threads_b;
Znak_his_prz[] NewThr_B = new Znak_his_prz[num_threads];
```

```

for (int i = 0; i < num_threads_b; i++) {
    if(num_threads_b == i-1) // zapewnienie pełnego histogramu
    {
        NewThr_B[i] = new Znak_his_prz(i, obraz_1 , krok * i, 94);
        NewThr_B[i].start();
    }
    else
    {
        NewThr_B[i] = new Znak_his_prz(i, obraz_1 , krok * i, krok * i +krok);
        NewThr_B[i].start();
    }
}

for (int i = 0; i < num_threads_b; i++) {
    try{
        NewThr_B[i].join();
        if (Thread.interrupted()) {
            throw new InterruptedException();
        }
    } catch (InterruptedException e){}
}

```

Ciało klasy extendującej **Thread**:

```

public /*static*/ class Znak_his_prz extends Thread {
    int i;
    int a;
    int b;
    Obraz obraz_1;
    public Znak_his_prz(int ia, Obraz ob, int aa, int ba)
    {
        i = ia;
        a = aa;
        b = ba;
        obraz_1 = ob;
    }
    @Override
    public void run()
    {
        obraz_1.calculate_histogram_par(a, b);
        obraz_1.print_histogram_par(a, b,i);
    }
}

```

Zamiast zliczania pojedynczego znaku każdy wątek zlicza przedział zapewniając pełne pokrycie histogramu.

Przykładowy histogram pojedynczego wątku

Watek 2: 7 - B

```
7      ##
8
9      ##
:
;
<      ###
=
>      ##
?
@      ###
A
```

Każdy wątek zlicza znak podany za pomocą **int** i wywołując odpowiednie metody przekazanego mu obiektu zawierający histogram i tablice do której zapisywane są wyniki.

```
Set number of threads char  
8  
  
Watek 0: !      #  
Watek 7: (      ##  
Watek 4: %      #  
Watek 6: '      #  
Watek 5: &      #  
Watek 3: $      #  
Watek 2: #      #  
Watek 1: "      #
```

Każdy wątek zlicza tylko jeden znak, brak pełnego histogramu w przypadku niewystarczającej liczby wątków.

laboratorium 7

Na laboratorium 7 zadaniem było napisanie programu obliczającego całkę metodą trapezów z wykorzystaniem interfejsu **ExecutorService** o zadanej liczbie wątków.

Wariant 1: Każdy wątek odpowiedzialny jest za jeden znak

Tworzenie puli wątków o zadanej liczbie i oczekiwanie na wyniki:

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

```
double dx = 0.0001;
double pi = Math.PI;
int ilosc = 50;
List<Future<Double>> flist = new ArrayList<>();
pi = pi/ (double) ilosc; // skalowanie zakresu dla kroku

for (int i = 0; i < ilosc; i++)
{
    Callable worker = new Calka_callable(i*pi, (i+1)*pi, dx);
    flist.add(executor.submit(worker));
}
executor.shutdown();
while (!executor.isTerminated()) {}

double wyn =0;
try{
    for (Future<Double> f : flist) {
        wyn += f.get();
    }
}
catch (Exception e) {}
System.out.println("Wynik: " + wyn);
}
```

Wyniki operacji są odbierane przez obiekty typu future i sumowane do wyniku.

Całka sin(x) z zakresu 0-Pi;

Wynik: 1.999999983369428