Tomasz Ligęza

Programowanie równoległe. Przetwarzanie równoległe i rozproszone.

Sprawozdanie z laboratorium 7.

Cel zajęć:

- nabycie umiejętności pisania programóœ w języku Java z wykorzystanie puli wątków

W ramach zajęć zrealizowałem następujące kroki:

1. Utworzyłem katalog roboczy lab_7,
2. Napisałem sekwencyjny program liczący całkę korzystając z dostarczonej klasy Calka_callable:

```
Calka_callable calka_callable = new Calka_callable(0, Math.PI, 0.001);
double result = calka_callable.compute();
System.out.println("Wynik: " + result);
```

3. Pobrałem paczkę java_executor_test.tgz,
4. Na podstawie przykładu z powyższej paczki napisałem program liczący całkę na zadanym przedziale wykorzystując interfejs ExecutorService oraz klasę Executors:

```
    private static final int NTHREADS = 10;
    private static final int NUMBER_OF_TASKS = 50;
    private static final double ACCURACY = 0.001;

    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(NTHREADS);
        List<Future<Double>> results = new ArrayList<>(NUMBER_OF_TASKS);
        double begin = 0.0;
        double end = Math.PI;
        double length = end - begin;
        double step = length/NUMBER_OF_TASKS;

        for (int i = 0; i < NUMBER_OF_TASKS; i++) {
            Calka_callable calka =
                new Calka_callable(step * (double)i,
                                   step * (double)(i+1),
                                   ACCURACY);
            Future<Double> future = executor.submit(calka);
            results.add(future);
        }
```

```
        double result = 0.0;
        for(var future : results) {
            try {
                result += future.get();
            } catch (ExecutionException | InterruptedException e) {
                e.printStackTrace();
            }
        }
        executor.shutdown();
        System.out.println("Wynik: " + result);
```

Zrzut ekranu z fragmentem wypisu zawierającego wynik:

```
xp = 3.0159289474462017, xk = 3.0787608005179976, N = 63
dx requested = 0.001, dx final = 9.973310011396183E-4
Creating an instance of Calka_callable
Calka czastkowa: 0.005912026623751145
xp = 3.0787608005179976, xk = 3.1415926535897936, N = 63
dx requested = 0.001, dx final = 9.973310011396183E-4
Calka czastkowa: 0.0019732714081657247
Calka czastkowa: 0.005912026623751152
Wynik: 1.99999983422181

Process finished with exit code 0
```

5. Stworzyłem kolejny katalog roboczy - lab_7_fork,
6. Uzupełniłem szkielet programu do sortowania oraz wykorzystując klasę ForkJoinPool uruchomiłem wielowątkowy program sortujący metodą merge-sort:

Metoda compute() z klasy DivideTask:

```
protected int[] compute() {
    if(arrayToDivide.length == 1)
        return arrayToDivide;
    DivideTask task1 = new DivideTask(Arrays.copyOfRange(arrayToDivide,
                                      0,
                                      arrayToDivide.length/2));
    DivideTask task2 = new DivideTask(Arrays.copyOfRange(arrayToDivide,
                                      arrayToDivide.length/2,
                                      arrayToDivide.length));
    task1.fork();
    task2.fork();
    int[] tab1 = task1.join();
    int[] tab2 = task2.join();
```

```
    int[] scal_tab = new int[arrayToDivide.length];
    scal_tab(tab1, tab2, scal_tab);
    return scal_tab;
}
```

Metoda main():

```
int[] numbers = {2, 4, 1, 2, -2, 9, -10};
DivideTask task = new DivideTask(numbers);
ForkJoinPool forkJoinPool = new ForkJoinPool();
forkJoinPool.execute(task);
System.out.println(Arrays.toString(task.join()));
```

Wywołanie programu:

```
/home/wigryz/.jdks/openjdk-17/bin/java -jav
[-10, -2, 1, 2, 2, 4, 9]

Process finished with exit code 0
```

7. Zmodyfikowałem program do obliczania histogramu tak, żeby korzystał z puli wątków:

```
int num_threads = Runtime.getRuntime().availableProcessors();
ExecutorService executor = Executors.newFixedThreadPool(num_threads);

int charsForThread = (int) Math.ceil((double)NUMBER_OF_CHARS/num_threads);
for (int i = 0; i < num_threads; i++) {
    int start = i * charsForThread;
    int end = Math.min(((i + 1) * charsForThread), NUMBER_OF_CHARS);
    executor.submit(new WatekDec(start, end, obraz_1));
}

executor.shutdown();
while(!executor.isTerminated()) {}

System.out.println("Czy oba histogramy sie zgadzaja? " +
                   obraz_1.checkBothHistogramsDec());
```

Zrzut ekranu z wykonania programu:

```
) 2
* 1
+ 1
, 0
Watek [pool-1-thread-1]: ! 2 ==
Watek [pool-1-thread-1]: " 2 ==
Watek [pool-1-thread-6]: + 1 =
Watek [pool-1-thread-6]: , 0
Watek [pool-1-thread-5]: ) 2 ==
Watek [pool-1-thread-5]: * 1 =
Watek [pool-1-thread-4]: ' 1 =
Watek [pool-1-thread-4]: ( 0
Watek [pool-1-thread-3]: % 1 =
Watek [pool-1-thread-3]: & 3 ===
Watek [pool-1-thread-2]: # 1 =
Watek [pool-1-thread-2]: $ 2 ==
Czy oba histogramy sie zgadzaja? true

Process finished with exit code 0
```

Wnioski:

Pule wątków bardzo ułatwiają pracę z wątkami. Dobrze prezentuje to ćwiczenie z sortowaniem przez scalanie. Rekursywne uruchamianie kolejnych wątków liczących podzadania byłoby dużo bardziej skomplikowane niż to z wykorzystaniem kasy ForkJoinPool.

Pule wątków umożliwiają nam pominięcie mozolnej pracy związanej z tworzeniem wątków, oczekiwaniem na zakończenie ich pracy oraz zarządzanie nimi. Dzięki temu możemy oddzielić tworzenie zadań od zarządzania wątkami.