Systemy Rozproszone - Ćwiczenie 2

1 Tworzenie wątku przez dziedziczenie po klasie Thread

Pierwszy sposób tworzenia wątku polega na stworzeniu klasy potomnej, dziedziczącej po klasie Thread. Zachowanie wątku określa się implementując metodę run() w klasie wątku. Wątek, po uprzednim stworzeniu, uruchamia się wywołując jego metodę start(). W konsekwencji system zacznie wykonywanie metody run() wątku.

```
public class HelloWorldThread extends Thread {
  public void run() {
    System.out.println("Hello World");
  }

  /**
   * Program demonstruje sposob tworzenia i
   * uruchomiania pojedynczego watku
   * @param args the command line arguments
   */
  public static void main(String[] args) {
    HelloWorldThread thread = new HelloWorldThread();
    thread.start();
  }
}
```

2 Przydzielanie wątkom czasu procesora

Poniższy program demonstruje dostęp do wspólnego zasobu (konsoli) równolegle przez 2 wątki i główny program. W main() tworzone i uruchamiane są 2 wątki, o nazwach "A" i "B". Każdy z wątków działa tak, że w pętli for wypisuje na standardowe wyjscie swoją nazwę. Dodatkowo metoda main() również wypisuje na standardow wyjście swoją nazwę ("M"). Zobacz w jaki sposób wątki i główny program rywalizują o dostęp do konsoli. Jak intensywny jest przeplot tego dostępu? Czy tego się spodziewałes?

```
public class HelloWorldThreadExt extends Thread {
```

```
  private String threadName;

  public HelloWorldThreadExt(String name) {
    this.threadName = name;
  }

  public void run() {
    for (int i=0; i<1000; i++) {
      System.out.print(this.threadName);
    }
  }

  /**
   * Program demonstruje uruchomienie 2 rownoleglych watkow i glownego programu
   */
  public static void main(String[] args) throws InterruptedException {
    HelloWorldThreadExt thread1 = new HelloWorldThreadExt("A");
    HelloWorldThreadExt thread2 = new HelloWorldThreadExt("B");
    thread1.start();
    thread2.start();
    for (int i=0; i<1000; i++) {
      System.out.print("M");
    }

  }
```

3 Tworzenie wątku przez implementację interfejsu Runnable

Innym sposobem tworzenia wątku jest implementacja interfejsu Runnable. W poniższym przykładzie tworzymy klasę HelloWorldRunnable, która dziedziczy po klasie Foo. Dodatkowo, klasa HelloWorldRunnable implementuje interfejs Runnable. Klasa implementująca ten interfejs musi zawierać implementację metody run(). Zwróć uwagę na sposób tworzenia wątku.

```
class Foo {
  private String name;

  public Foo(String name) {
    this.name = name;
  }

  public String get_name() {
    return this.name;
  }
```

```
}

public class HelloWorldRunnable extends Foo implements Runnable {

  public HelloWorldRunnable(String name) {
    super(name);
  }

  public void run() {
    System.out.println("Thread "+this.get_name()+" is running");
  }

  /**
   * Program demonstruje utworzenie klasy z interfejsem Runnable
   * (w celu umozliwienia dziedziczenia) i uruchomienie takiego obiektu
   * jako watku
   */
  public static void main(String[] args) throws InterruptedException {
    HelloWorldRunnable runnable = new HelloWorldRunnable("runnable");
    Thread thread = new Thread(runnable);
    thread.start();
  }
}
```

4 Synchronizacja wątków - join()

Wywołanie metoda x.join() wątku x, pozwala na wstrzymanie wykonywania bieżącego wątku do czasu aż wątek x sie zakończy. Ilustruje to poniższy przykład, w którym wątek główny tworzy 2 wątki poboczne - zadaniem pierwszego jest policzenie sumy elementów nieparzystych w tablicy, zadaniem drugiego jest zliczenie elementów nieparzystych w tablicy. Wątek główny uruchamia 2 wątki poboczne i czeka aż skończą pracę. Następnie oblicza średnią na podstawie wyników obliczonych przez wątki poboczne.

```
class OddCountThread extends Thread {
    protected int[] numbers;
    protected int result=0;

    public OddCountThread(int[] numbers) {
        this.numbers = numbers;
    }

    public boolean is_odd(int element) {
        return element % 2 == 1;
    }
```

```
    public void run() {
        for(int element : numbers) {
            if (is_odd(element))
                result++;
        }
    }

    public int get_result() {
        return result;
    }
}

class OddSumThread extends OddCountThread {
    public OddSumThread(int[] numbers) {
        super(numbers);
    }

    @Override
    public void run() {
        for(int element : numbers) {
            if (is_odd(element))
                result+=element;
        }
    }
}

public class ThreadedAverage {

    public static void main(String[] args) {
        int[] numbers = {1, 8, 5, 3, 2, 5, 99, 6};

        OddCountThread count_job = new OddCountThread(numbers);
        OddSumThread sum_job = new OddSumThread(numbers);
        sum_job.start();
        count_job.start();

        try {
            sum_job.join();
            count_job.join();
            float oddAverage = (float)sum_job.get_result() / (float)count_job.ge
            System.out.println("Srednia z elementow nieparzystych to " + oddAver
        }
        catch (InterruptedException e) {
            System.err.println(e);
        }
```

```
    }
}
```

Następny przykład jest trochę bardziej skomplikowany. Przeanalizuj jego działanie.

```
public class CalcSum extends Thread {
  int[] data;
  int first, last;
  int result;

  public int get_result() {
    return this.result;
  }

  public CalcSum(int[] data) {
    this.result = 0;
    this.data = data;
    this.first = 0;
    this.last = data.length-1;
  }

  public CalcSum(int[] data, int first, int last) {
    this.result = 0;
    this.data = data;
    this.first = first;
    this.last = last;
  }

  public void run() {
    if (first==last) {
      result = data[first];
    }
    else {
      int middle = (first+last)/2;
      CalcSum c1 = new CalcSum(data, first, middle);
      CalcSum c2 = new CalcSum(data, middle+1, last);
      c1.start();
      c2.start();
      try {
        c1.join();
        c2.join();
      } catch (InterruptedException e) {};
      result = c1.get_result() + c2.get_result();
    }

  }
```

```
  /**
   * Program oblicza sume elementow w tablicy w sposob rekurencyjny,
   * przy pomocy watkow.
   * Program demonstruje w jaki sposob watek moze czekac na zakonczenie innych
   * watkow.
   */
  public static void main(String[] args) {
    int[] data = {1, 1, 2, 6, 24, 120, 720, 5040};
    CalcSum c = new CalcSum(data);
    c.start();

    // to zadziala niepoprawnie
    System.out.println("Sum is " + c.get_result());

    // to zadziala poprawnie
    try {
      c.join();
    } catch (InterruptedException e) {};
    System.out.println("Sum is " + c.get_result());
  }
}
```

5 Zadania

5.1 Suma wielkiej tablicy

Zmodyfikuj poniższy program tak, żeby suma tangensów była liczona w sposób wielowątkowy. Zmień program tak, żeby łatwo można było zmienić liczbę wątków użytych do obliczenia sumy. Sprawdź eksperymentalnie jaka jest zależność między liczbą wątków a czasem wykonania programu. Przeprowadź eksperymenty dla tablic o różnych wielkościach (100, 10000, 1000000, ...) i dla różnej wielkości wątków (1,2,4,8,16,32).

```
public class SumOfTangents {
    public static void main(String[] args) {
        double[] array = new double[100 * 1000000];

        Random r = new Random();

        for (int i=0; i<array.length; i++) {
            array[i] = r.nextDouble();
        }

        long startTime = System.currentTimeMillis();
```

```
        double total = 0;
        for (int i=0; i<array.length; i++) {
            total += Math.tan(array[i]);
        }
        long stopTime = System.currentTimeMillis();
        System.out.println("Total is: " + total);
        System.out.println("Elapsed time: " + (stopTime - startTime) + " milisec
    }
}
```

5.2 Wyścig wątków

Załóżmy, że stworzymy np. 10 wątków, z których każdy wykonywać bedzie te same obliczenia. Czy wątki te zakończą obliczenia w takiej samej kolejności jak zostały uruchomione? Czy tak kolejność bedzie taka sama za każdym uruchomieniem programu? Sprawdźmy to eksperymentalnie. Stwórz klase Zawodnik, która ma być wątkiem wykonującym pewne obliczenia (np. zapełnij 1000 elementową tablice klasy Zawodnik kolejnymi wartosciam funcji tangens). Stwórz klase Wyscig, która stworzy i uruchomi wątki klasy Zawodnik. Każdy zawodnik po zakończeniu obliczeń powinien zgłosić ten fakt (np. wypisując komunikat na konsoli). Przeprowadź eksperyment dla różnej ilości wątków i różnego poziomu skomplikowania metody run(). Czy zauważasz jakieś prawidłowości? Rozbuduj program o klase Tablica wyników. Klasa wyścig powinna przydzielić każdemu zawodnikowi numer, a po zakończeniu pracy przez zawodnika, powinien on zgłosić swój numer do tablicy wyników (wywołując odpowiednią metode klasy Tablica), a nastepnie tablica wypisze stosowna informacje na konsoli.

5.3 Sztafeta wątków

Zmodyfikujmy wyścigi w nastepujacy sposob. Wyścigi rozgrywane bedą w sztafecie 10 zespołów po 4 wątki. Zawodnicy zostana podzieleni na zespoły (klasa Zespół), i to teraz zespoły, a nie zawodnicy, biorą udział w wyścigu. Każdy zawodnik zna swój numer startowy i numer zespołu do której należy. Klasa Zespół ma być odpowiedzialna za uruchamianie kolejnych wątków należących do jedego zespołu ale w taki sposób, żeby tylko jeden zawodnik w zespole był uruchomiony jednocześnie (do tego celu należy użyć metody `join()`). Klasa zespół powinna informować o rozpoczeciu/zakończeniu wyścigu przez zawodnika i o zakończeniu sztafety przez ten zespół. Zmodyfikuj program tak, żeby to zawodnik czekał na zokończenie pracy przez poprzedniego zawodnika.