Задание:

Създаване на многонишкова програма за сумирането на дължимите суми от фактури, намиращи се в голям файл, представляващ единствен ресурс

Десислава Милушева   471219007
Николай Каймакански 471219072
ИСН 3 курс 77 група

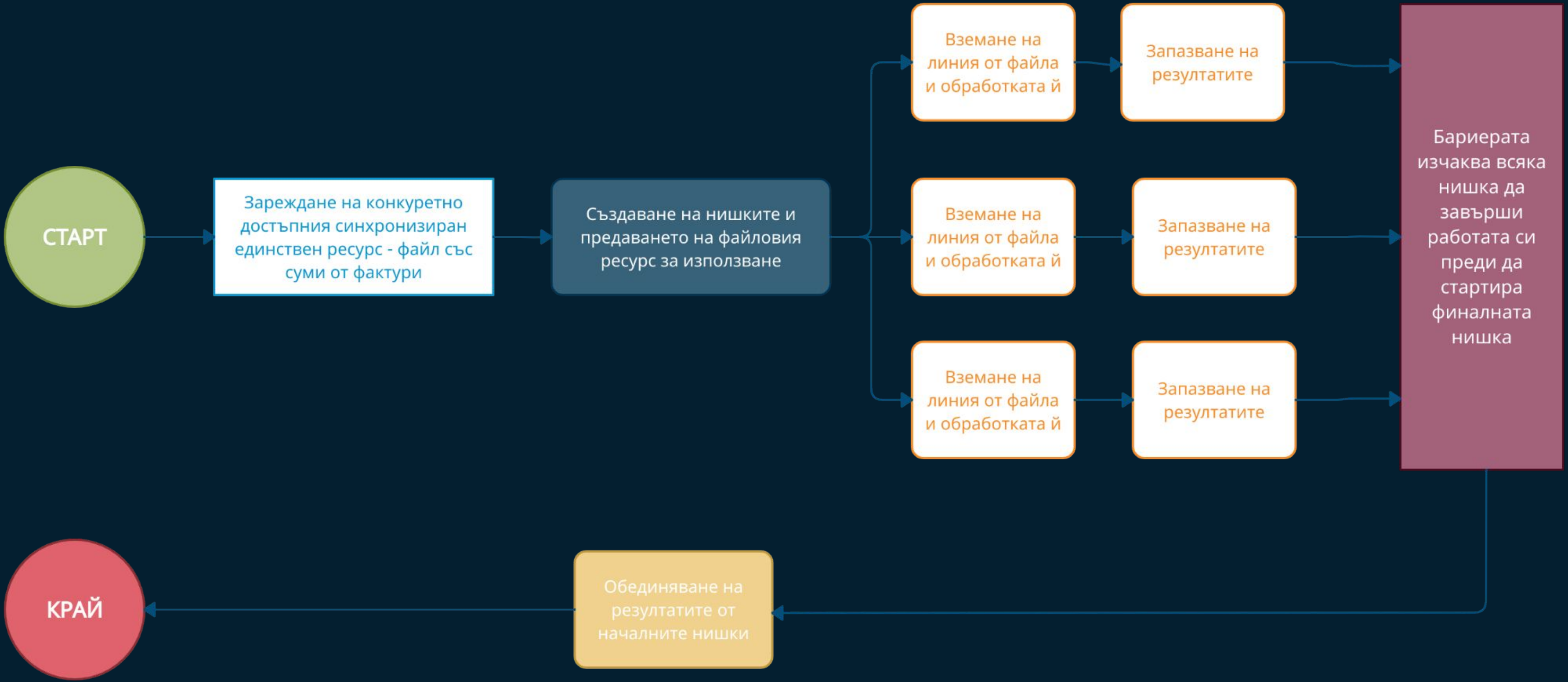
Съдържание:

1. Описание на заданието

Да се създаде многонишкова програма за сумирането на дължимите суми от фактури, намиращи се в голям файл, представляващ единствен ресурс.

Резултатите от извършването на сумирането от всяка отделна нишка трябва да бъдат сумирани и изведени, след като всяка от нишките е завършила изпълнението си.

2. Модел на паралелните изчисления


СТАРТ
Зареждане на конкуретно достъпния синхронизиран единствен ресурс - файл със суми от фактури
Създаване на нишките и предаването на файловия ресурс за използване
Вземане на линия от файла и обработката й
Запазване на резултатите
Вземане на линия от файла и обработката й
Запазване на резултатите
Вземане на линия от файла и обработката й
Запазване на резултатите
Бариерата изчаква всяка нишка да завърши работата си преди да стартира финалната нишка
Обединяване на резултатите от началните нишки
КРАЙ

3. Имплементация на Java

https://github.com/desi109/invoices-sum-calculator---multithreading-java-app

```java
public class InvoicesSumCalculatorMultithreading  {

   private static final  String FILE_PATH = new File(Paths.get(".").toString(),  "resources/invoices.csv" ).getAbsolutePath();
   private static final int  NUM_THREADS = 3;
   private static CyclicBarrier  barrier;
   private static List<Float> results = new ArrayList<>(NUM_THREADS);

   private static List<Thread> threads = new ArrayList<>(NUM_THREADS);

   public static void  main(String[] args) {

       // 1. Start the program and the main thread
       try {
           // initialize a CyclicBarrier to wait for all FileLineProcessingThread to finish, before start the ResultConsolidationThread
          barrier = new CyclicBarrier(NUM_THREADS, new ResultFinalizationThread( results));
           long beforeUsedMemory  = Runtime.getRuntime().totalMemory() -  Runtime.getRuntime().freeMemory();

           // 2. Load and process the file invoices.csv
          CsvFileReader csvFileReader  = new CsvFileReader(FILE_PATH);
           Watcher watcher  = new Watcher();
           watcher.startTimeNanos();
           processPostsByLineMultithreading(csvFileReader);
           watcher.endTimeNanos();

           long afterUsedMemory  = Runtime.getRuntime().totalMemory() -  Runtime.getRuntime().freeMemory();

           System.out.println("Reading took: "  + watcher.timeMillis() +  " ms");
           System.out.println("Memory used for the for the whole multithreading program: "   +  ((afterUsedMemory  - beforeUsedMemory) / 1024.0) +
" MB");

       } catch  (FileNotFoundException  ex) {
           System.out.println(FILE_PATH + " does not exists!");
       }
   }

   private static void processPostsByLineMultithreading (CsvFileReader  csvFileReader) {
       for (int i = 1; i <= NUM_THREADS; ++i) {
           FileLineProcessingThread thread  = new FileLineProcessingThread( "Thread #"  + i, csvFileReader,  barrier, results);
           thread.start();
           threads.add(thread);
       }
```

```java
public class CsvFileReader implements AutoCloseable {

   private FileReader fr = null;
   private StringBuilder sb = new StringBuilder();
   private int i;

   public CsvFileReader (String fileLocation) throws FileNotFoundException {
       fr = new FileReader(fileLocation);
   }

   public synchronized List<String> getCsvLine () throws IOException {
       sb.setLength(0);
       List<String> fileLine = new ArrayList<>();

       // read every line, split its elements by comma, and put them iн fileLine ArrayList
      while ((i = fr.read()) != -1) {
           char c = (char) i;

           if (c == 10) {  // 10 -> NEW LINE (\n)
               for (String element : sb.toString().split(",")) {
                    fileLine.add(element);
               }
               sb.setLength(0);
               return fileLine;
           } else {
               sb.append(c);
           }
       }

       if (sb.length() != 0) {
           for (String element : sb.toString().split(",")) {
               fileLine.add(element);
           }
           sb.setLength(0);
           return fileLine;
       }
       return null;
   }

   @Override
   public void close() { }
public class FileLineProcessingThread extends Thread {
   private String threadName;
   private CsvFileReader csvFileReader;
   private CyclicBarrier barrier;
   private List<Float> results;

   public FileLineProcessingThread(String threadName, CsvFileReader csvFileReader, CyclicBarrier barrier, List<Float> results) {
       this.threadName = threadName;
       this.csvFileReader = csvFileReader;
       this.barrier = barrier;
       this.results = results;
   }

   @Override
   public void run() {
       Watcher watcher = new Watcher();
       watcher.startTimeNanos();
       int fileLinesSize = 0;
       float sumOfAllInvoicesForCurrentThread = 0.0f;
       List<String> fileLine = new ArrayList<>();

       try {
           fileLine = csvFileReader.getCsvLine();

           while (fileLine != null) {
               ++fileLinesSize;

               // check if the sixth element is numeric (the invoice amount)
               if (isNumeric(fileLine.get(5))) {
                   float invoiceAmount = Float.parseFloat(fileLine.get(5));
                   if (isNumeric(fileLine.get(4))){
                       float invoiceQuantity = Float.parseFloat(fileLine.get(4));
                       sumOfAllInvoicesForCurrentThread += invoiceAmount * invoiceQuantity;
                   } else {
                       sumOfAllInvoicesForCurrentThread += invoiceAmount;
                   }

                   //simulate more complicated computational work
                   // Thread.sleep(1);
               } else {
                   if (fileLine.size() < 5) {
                       String lineContent = "[";
                       int elementNumber = 0;
                       for (String element : fileLine) {
                           if ((fileLine.size() - 1) == elementNumber) {
                               lineContent += element.trim() + "]";
                           } else {
                               lineContent += element.trim() + ", ";
                           }
                           elementNumber++;
                       }
```

```java
                        System.out.println("Warning: inconsistent line: "+ fileLinesSize +"! Content: " + lineContent);
                        continue;
                    }
                }

                fileLine =csvFileReader.getCsvLine();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }

        results.add(sumOfAllInvoicesForCurrentThread);

        watcher.endTimeNanos();
        System.out.println("Execution time of thread "+ threadName + ": " + watcher.timeMillis() +" ms");
        System.out.println("Sum of all invoices of thread "+ threadName + ": " + sumOfAllInvoicesForCurrentThread);
        System.out.println("File lines size processed by thread "+ threadName + ": " + fileLinesSize);

        try {
            // the CyclicBarrier will wait for all FileLineProcessingThread to finish, before start the ResultConsolidationThread
            barrier.await();
        } catch (InterruptedException| BrokenBarrierException e) {
            e.printStackTrace();
        }
    }

    private static boolean isNumeric(String strNum) {
        if (strNum == null) {
            return false;
        }
        try {
            float f = Float.parseFloat(strNum);
        } catch (NumberFormatException nfe) {
            return false;
        }
        return true;
    }
}
```

```java
public class ResultFinalizationThread extends Thread {
   private List<Float> results;

   public ResultFinalizationThread (List<Float> results) {
        this.results = results;
   }

   @Override
   public void run() {
        System.out.println("Result Finalization Thread started!" );
        float sum = 0.0f;

        for (Float result : results) {
            sum += result;
        }

        System.out.println("Invoices sum: " + sum);
   }
}
```

```java
public class Watcher {
   private long startTime = -1;

   public void startTimeNanos () {
        this.startTime = System.nanoTime();
   }

   public long endTimeNanos () {
        return System.nanoTime() - this.startTime;
   }

   public double timeMillis () {
        return this.endTimeNanos() / 1000000.0;
   }
}
```

// InvoicesSumCalculatorSingleThreaded class

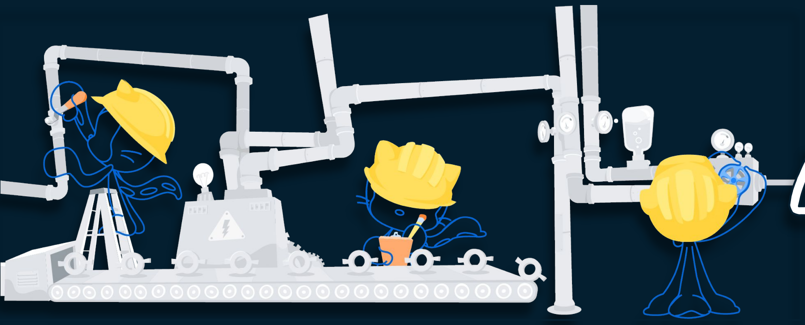
```java
public class InvoicesSumCalculatorSingleThreaded {

   private static final String FILE_PATH = new File(Paths.get(".").toString(), "resources/invoices.csv").getAbsolutePath();

   public static void main(String[] args) {

       // 1. Start the program and the main thread
       try {
           long beforeUsedMemory = Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory();

           // 2. Load and process the file invoices.csv
           CsvFileReader csvFileReader = new CsvFileReader(FILE_PATH);
           Watcher watcher = new Watcher();
           watcher.startTimeNanos();
           processPostsByLineSingleThreaded(csvFileReader);
           watcher.endTimeNanos();

           long afterUsedMemory = Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory();

           System.out.println("Reading took: " + watcher.timeMillis() + " ms");
           System.out.println("Memory used from a single thread: " + ((afterUsedMemory - beforeUsedMemory) / 1024.0) + " MB");

       } catch (FileNotFoundException ex) {
           System.out.println(FILE_PATH + " does not exists!");
       }
   }

   private static void processPostsByLineSingleThreaded(CsvFileReader csvFileReader) {
       Watcher watcher = new Watcher();
       watcher.startTimeNanos();
       int fileLinesSize = 0;
       float sumOfAllInvoicesForCurrentThread = 0.0f;
       List<String> fileLine = new ArrayList<>();

       try {
           fileLine = csvFileReader.getCsvLine();

           while (fileLine != null) {
               ++fileLinesSize;

               // check if the sixth element is numeric (the invoice amount)
               if (isNumeric(fileLine.get(5))) {
                   float invoiceAmount = Float.parseFloat(fileLine.get(5));
                   if (isNumeric(fileLine.get(4))) {
                       float invoiceQuantity = Float.parseFloat(fileLine.get(4));
                       sumOfAllInvoicesForCurrentThread += invoiceAmount * invoiceQuantity;
                   } else {
```

```java
     sumOfAllInvoicesForCurrentThread +=invoiceAmount;
                   }
                  //simulate more complicated computational work
                   // Thread.sleep(1);
              } else {
                  if (fileLine.size() < 5) {
                      String lineContent = "[";
                      int elementNumber = 0;
                      for (String element : fileLine) {
                          if ((fileLine.size() - 1) == elementNumber) {
                               lineContent +=element.trim() + "]";
                           }else {
                               lineContent +=element.trim() + ", ";
                           }
                           elementNumber++;
                       }

                      System.out.println("Warning: inconsistent line: "+ fileLinesSize +"! Content: " + lineContent);
                      continue;
                   }
              }

              fileLine = csvFileReader.getCsvLine();
          }
      } catch (IOException e) {
          e.printStackTrace();
      }

      watcher.endTimeNanos();
      System.out.println("Execution time for a single thread: "+ watcher.timeMillis() + " ms");
      System.out.println("File lines size processed by a single thread: "+ fileLinesSize);
      System.out.println("Invoices sum:  " + sumOfAllInvoicesForCurrentThread);
  }

  private static booleanisNumeric(String strNum) {
      if (strNum == null) {
          return false;
      }
      try {
          float f = Float.parseFloat(strNum);
      } catch (NumberFormatExceptionnfe) {
          return false;
      }
      return true;
  }
}
```

4. Результати

```
1 thread - InvoicesSumCalculatorSingleThreaded.java
----------------------------------------------------
Execution time for a single thread: 12146.699535 ms
File lines size processed by a single thread: 10000
Invoices sum:  2656788.2
Reading took: 12176.839503 ms
Memory used from a single thread: 43302.390625 MB
```

```
4 threads - InvoicesSumCalculatorMultithreading.java
-----------------------------------------------------
Reading took: 16.212648 ms
Memory used for the for the whole multithreading program: 1064.890625 MB
Execution time of thread Thread #1: 2932.733991 ms
Execution time of thread Thread #3: 2931.733848 ms
Execution time of thread Thread #4: 2931.502864 ms
Execution time of thread Thread #2: 2932.124885 ms
Sum of all invoices of thread Thread #1: 661921.8
Sum of all invoices of thread Thread #2: 662503.7
Sum of all invoices of thread Thread #4: 663492.1
Sum of all invoices of thread Thread #3: 668875.1
File lines size processed by thread Thread #3: 2500
File lines size processed by thread Thread #2: 2501
File lines size processed by thread Thread #1: 2500
File lines size processed by thread Thread #4: 2499
Result Finalization Thread started!
Invoices sum: 2656788.2
```



Благодаря за вниманието!