Object Orientation – Spring 2018 **Assignment 13**

Goals

This exercise consists of making two programs with multiple threads. Use the approach for designing concurrent programs outlined in the lectures. After this exercise you should be able to:

- create threads to execute parts of a computation;
- synchronize threads after termination and combine their result;
- speed-up a Java program by using multiple cores.

1 Find File

The Java class `java.io.File` represents all files and directories on your computer. We will use this class to locate a file with a given `name` somewhere in the nested directory system. The program below prints the full path of all files with that name. `FileFinder` is a class that executes this search recursively. The constructor sets the initial directory if it exists and throws an exception otherwise. The recursive search by the private method `find` is initiated by a call to the public method `findFile` of a `FileFinder` object.

```
public class FileFinder {
    private final File rootDir;

    public FileFinder(String root) throws IOException {
        rootDir = new File(root);
        if (! (rootDir.exists() && rootDir.isDirectory())) {
            throw new IOException(root + " is not a directory.");
        }
    }

    public void findFile(String file) {
        find(rootDir, file);
    }

    private void find (File rootDir, String fileName) {
        File [] files = rootDir.listFiles();
        if (files != null) {
            for (File file: files) {
                if (file.getName().equals(fileName)) {
                    System.out.println("Found at: " + file.getAbsolutePath());
                } else if (file.isDirectory()) {
                    find(file, fileName);
                }
            }
        }
    }
}
```

The class `FileFinderTest` searches for the file named `FileFinder.java` in on disk `C`.

```
public class FileFinderTest {

    public static void main(String[] args) {
        fileFinderTest();
   }

    public static void fileFinderTest() {
        try {
            String goal = "FileFinder.java";
            String root = "C:\\";
            FileFinder ff = new FileFinder(root);
            ff.findFile(goal);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Since there can be many directories and files, such a search can take a lot of time. Your task is to speed-up the search for a file by creating a new searching thread for each directory encountered. In the code above you have to replace the recursive call `find(file, fileName)` by the creation of a thread that tries to find the file `fileName` in the directory `file`.

This way, your program can create a huge numbers of threads. Since each thread is an Java object, it occupies some memory space in the heap. With a huge number of threads the heap space of your program might become exhausted; Java will generate an exception when there is not enough memory left to create a thread. You can limit the size of the search by choosing a better starting directory. Moreover, you can give your Java program more executing space to accommodate more threads. By extending the merge space of the Java virtual machine it can contain more threads before it throws an exception indicating that there is no space to create an additional thread. You can set the maximum heap space of your Java virtual machine to 2048 MB by the option `-Xmx2048m` in the properties of your project. In NetBeans, this option can be found in the `Run` section of the project options. Depending on your installation, the current maximum heap space might be smaller or larger than the size in this example.

2 Merge Sort

Merge sort is sorting algorithm with time complexity $O(N \log N)$. In this algorithm the array to be sorted is split into two parts of equal size. These parts are sorted recursively. Next, these sorted arrays are merged into one sorted array. To make this final step easy, the sub-arrays in the first step of the algorithm are copies of the original array. Since the arrays to be merged are sorted, merging them into a single sorted array is easy. In contrast to the sorting algorithms discussed in the courses *Programming for AI* and *imperatief programmeren*, this implementation of merge sort is not *in situ*; merge sort does not run in constant space. The arrays `firstHalf` and `secondHalf` do require additional temporary memory. The class `MergeSort` provides an implementation of this algorithm.

```
public class MergeSort {
  /**
   * sort the given array in O(N log N) time
   * The array is split into two parts of equal size.
   * These parts are sorted recursively and merged.
   * @param array
   */
  public static void sort(int[] array) {
    if (array.length > 1) {
      int[] firstHalf  = Arrays.copyOf(array, array.length / 2);
      int[] secondHalf = Arrays.copyOfRange(array, array.length / 2, array.length
          );
      sort(firstHalf);
      sort(secondHalf);
      merge(firstHalf, secondHalf, array);
    }
  }

  /**
   * merge two sorted arrays: time O(N)
   * @param part1 a sorted array
   * @param part2 a sorted array
   * @param dest  destination, length must be >= part1.length + part2.length
   */
  public static void merge(int[] part1, int[] part2, int[] dest) {
    int part1Index = 0, part2Index = 0, destIndex = 0;
    while (part1Index < part1.length && part2Index < part2.length) {
      if (part1[part1Index] < part2[part2Index])
        dest[destIndex ++] = part1[part1Index ++];
      else
        dest[destIndex ++] = part2[part2Index ++];
    }
    // copy elements when at most one of the parts contains elements
    while (part1Index < part1.length)
      dest[destIndex ++] = part1[part1Index ++];
    while (part2Index < part2.length)
      dest[destIndex ++] = part2[part2Index ++];
  }
}
```

This algorithm is very suited for parallelization; the arrays `firstHalf` and `secondHalf` have the same size and can be sorted independently. Since creating a thread to sort such an array is also work, you should only create such a thread when the size of the array is bigger than some threshold value (e.g. 1000). When the size of the array is below the threshold, the sequential version of the sorting algorithm is used.

This idea of parallelization of large problems and solving small problems sequentially is an instance of a general *divide and conquer pattern* for recursive problems:

```
if (size < threshold) {
    sequential_solution;
} else {
    divide_in_nonoverlapping_subproblems;
    solve_subproblems_in_parallel;
    combine_results;
}
```

Your task is to make a parallel version of the merge sort algorithm where sub-arrays are sorted in parallel by different threads. We can only start merging the sub-arrays when their sorting threads are finished. Hence, your program needs an explicit `join`.

In order to measure the effect of parallelization you apply the sequential and parallel version of merge sort to a long array of pseudo random integers, say 10,000,000 elements. You can obtain the current time in milliseconds by `System.currentTimeMillis()`. The Java runtime method `Runtime.getRuntime().availableProcessors()` returns the number of processors or cores available on your machine. By adjusting the threshold of your program, it should be possible to obtain a speed-up of 2 at a 4-core processor. Include the timing results and the number of cores of your computer as a comment in your program.

Deadline

Sunday May 28, 23:59.