ECE 360P: Concurrent and Distributed Systems
Assignment 1

Instructor: Professor Vijay K. Garg

Deadline: 2:00pm Jan. 31^{st}, 2023

This homework contains a programming part (Q1-Q2) and a theory part (Q3-Q4). The theory part should be either typed and submitted as a pdf file or written on a paper, scanned and then submitted online as a pdf file. The file should be named [Theory-EID1_EID2].pdf. This file should be submiited in the directory that has the source code for the programming part. The source code (Java files) of the programming part and the pdf file for the non-programming part must be uploaded through the canvas before the deadline (i.e., 2:00pm on Jan. 31^{st}). There will be a late penalty of 10% of the total grade for each day the assignment is late. The assignment should be done in teams of two. You should use the templates downloaded from the course github (https://github.com/vijaygarg1/ECE-360P.git). You should not change the file names and function signatures. In addition, you should not use package for encapsulation. The zip folder you submit should contain only java files with no sub-folders. The source files for both questions should be in the same folder. Please zip and name the source code as [EID1_EID2].zip.

1. **(35 points)** In this question, we will write two Java classes `RunnablePSort` and `ForkJoinPSort` which allow for parallel sorting of an array of integers. The classes both provide the following `static` method:

   ```
   public static void parallelSort(int[] A, int begin, int end, boolean increasing) {
       // your implementation goes here.
   }
   ```

   For both classes, the `parallelSort` method will employ the *QuickSort* algorithm. Quicksort is a divide and conquer algorithm. In the first step, the input array is divided into two sub-arrays, which we call the left and right array. Recursive calls are made on the left and right sub-arrays, which are divided and sorted using the same strategy. Once the input array size is less than or equal to 16, the recursive division stops and the simple sequential insert sort is used for sorting. In your implementation, in each step tasks for recursing on left and right sub-arrays should be scheduled to operate in parallel, which in turn allows for parallel sorting. The `ForkJoinPSort` class should use `ForkJoinPool` for the parallel implementation, whereas the `RunnablePSort` will use Java Runnables. The input array `A` is also the output array, which will make your job easier. The range to be sorted extends from index `begin`, inclusive, to index `end`, exclusive. In other words, the range to be sorted is empty when `begin == end`. Moreover, the method signature includes the boolean flag `increasing` which directs the method to sort in increasing order when `true` and decreasing order when `false`. To simplify the problem, you can assume that the size of array `A` does not exceed 10,000.

2. **(25 points)** Write a Java class that performs a parallel merge of two arrays of integers. It provides the following `static` method:

```
public class PMerge {
    public static void parallelMerge(int[] A, int[] B, int[] C, int numThreads) {
        // arrays A and B are sorted in the ascending order
        // These arrays may have different sizes.
        // array C is the merged array sorted in the descending order
        // your implementation goes here.
    }
}
```

This method uses as many threads as specified by `numThreads`. It merges sorted arrays `A` and `B` into the array `C` sorted in the reverse order. You may assume that both arrays `A` and `B` are sorted and that there are no duplicates in `A` and `B`. (i.e., all elements in `A` are unique, all elements in `B` are unique; however, `A` and `B` may have elements in common. The elements that are common must appear in C as duplicates.

3. **(25 points)** Show that any of the following modifications to Peterson's algorithm makes it incorrect:

   a) A process in Peterson's algorithm sets the *turn* variable to itself instead of setting it to the other process. The remaining algorithm stays the same.

   b) A process sets the *turn* variable before setting the *wantCS* variable.

4. **(15 points)** Prove that Peterson's algorithm is free from starvation.