Programming Assignment III

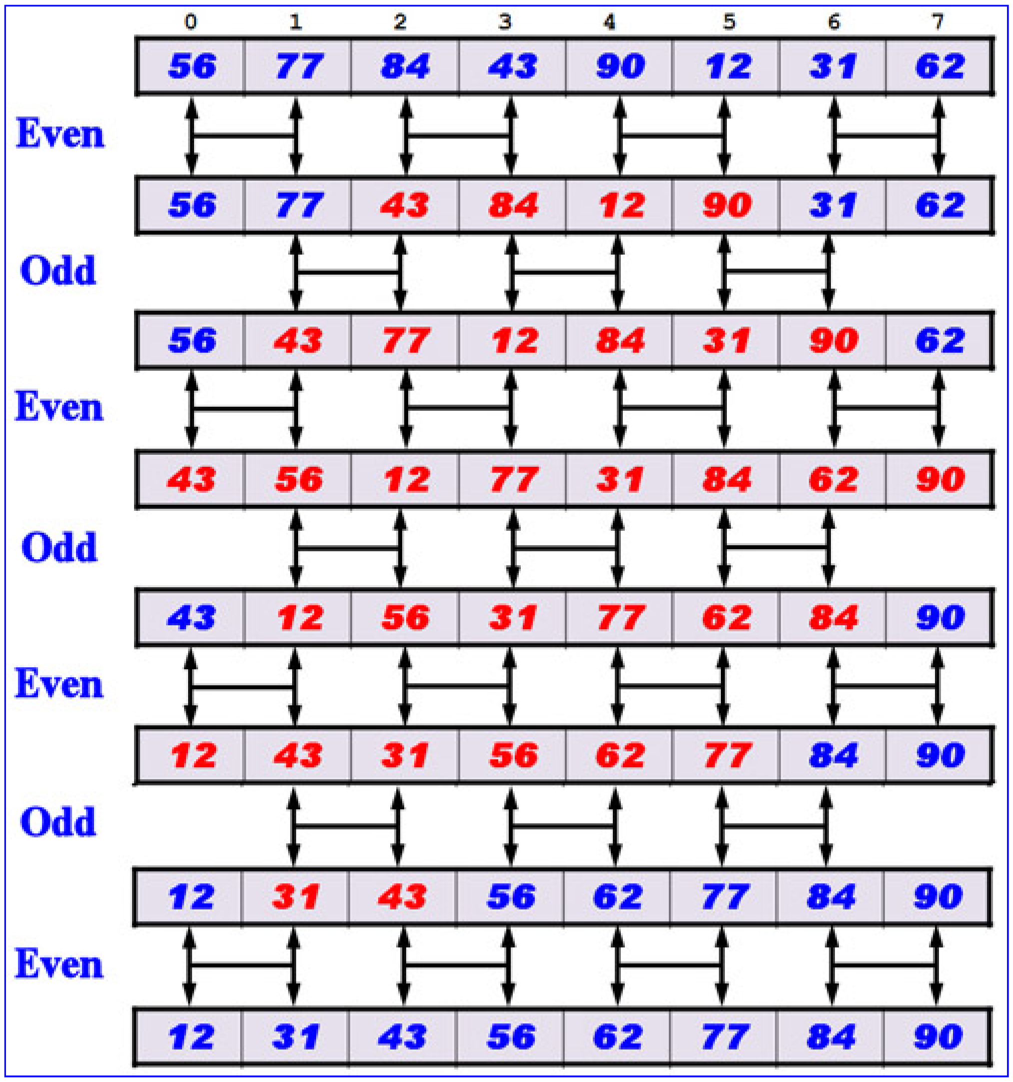
Due on Wednesday, November 3rd, 2021 @ 11:59pm

50 points

This is a warm-up simple multithreaded programming assignment using **ThreadMentor**.

The Even-Odd Sorting Algorithm

Let us start talking about what the Even-Odd sorting algorithm is. Consider a case of eight numbers: 56, 77, 84, 43, 90, 12, 31 and 32 as shown in the diagram below. An **Even** pass means we compare two adjacent array elements such as **`x[2i]`** and **`x[2i+1]`** (*i.e.*, **`x[0]`** and **`x[1]`**, **`x[2]`** and **`x[3]`**, etc.). If they are out of order, swap them. On the other hand, an **Odd** pass means we compare two adjacent array elements such as **`x[2i+1]`** and **`x[2i+2]`** (*i.e.*, **`x[1]`** and **`x[2]`**, **`x[3]`** and **`x[4]`**, etc.). If they are out of order, swap them. In the diagram below, an even pass compares 56-77, 84-43, 90-12 and 31-62, and the new array is 56, 77, 43, 84, 12, 90, 31 and 62. In the diagram, we use red color to indicate the two adjacent swapped numbers. Then, we try an odd pass, which compares 77-43, 84-12, 90-31, and the new array is 56, 43, 77, 12, 84, 31, 90 and 62. Note that the last element in the array (*e.g.*, 62) may not be able to form a pair. In this case, we just leave it alone. This an even pass followed by an odd pass process continues until there is no swaps. Then, we have a sorted array. We use a flag, initially **`FALSE`**, and set it to **`TRUE`** if there is a pair of numbers gets swapped. At the end of each even pass and odd pass we check the flag, and if no swpas were found in **both** passes we stop because the array has been sorted.



It is very easy to write a sequential program for the even-odd sort:

```
/* This is the compare and swap function */
/*     Pass = 0 ==> even pass            */
/*     Pass = 1 ==> odd pass             */

int  Sort(int Pass, int x[], int n)
{
     int i, temp, Swapped = FALSE;

     for (i = 1 + Pass; i < n, i += 2)
          if (x[i-1] > x[i]) {
               temp   = x[i-1];
               x[i-1] = x[i];
               x[i]   = temp;
               Swapped = TRUE;
          }
     }
     return Swapped;
}

/* This is the even-odd sorting function        */

#define Even 0  /* even pass starts from 1+Even */
#define Odd  1  /* odd pass starts from 1+Odd   */

int  Swapped, i, n;
```

```
Swapped = TRUE;
while (Swapped) {
     Swapped = Sort(Even, x, n);             /* Even Pass  */
     Swapped = Swapped || Sort(Odd, x, n);   /* Odd Pass   */
}
```

In the **`Sort()`** function above, an even pass compares **`x[1]`** and **`x[0]`**, **`x[3]`** and **`x[2]`**, **`x[5]`** and **`x[4]`**, etc. For an odd phase, it compares **`x[2]`** and **`x[1]`**, **`x[4]`** and **`x[3]`**, **`x[6]`** and **`x[5]`**, etc. Therefore, the initial indices are 1 and 2 for an even pass and odd pass respectively. The last index of both pass is at most **`n`**-1.

The complexity of the Even-Odd sort is rather high. Each iteration executes one even pass and one odd pass, each of which requires at most $n/2$ comparisons. Hence, the number of comparisons in each iteration is n, and the total number of comparisons to sort the array is $O(n^2)$ because n iterations are needed.

Making the Even-Odd Sort Concurrent!

The Even-Odd sort can easily be converted to a concurrent version. Note that all of the $n/2$ comparisons in each pass are ***independent*** of each other, and can be executed concurrently. Therefore, for each even or odd pass, we may create $n/2$ threads, each of which only compares two adjacent entries of the array and swaps them if needed. Once a pass completes, we may either kill all threads and create another $n/2$ new threads to perform the next pass. Or, if you are smart, you could also use the original $n/2$ threads to sort a different pair of adjacent elements; however, you have to be very careful so that race conditions will not occur. So, let us go for the dumb way to save our time.

The Algorithm:

From the above, we are able to quickly develop an algorithm to do a concurrent Even-Odd sort. It is summarized as follows:

1. Suppose the input array **`x[*]`** has n numbers.
2. Use a **`while`** loop that iterates at most n times as shown earlier. In each iteration, do the following until no swaps occur.
    - **Even Pass**:
        - Create at most $n/2$ threads: T_1, T_3, T_5, etc.
        - Thread T_k compares **`x[k-1]`** and **`x[k]`**. If **`x[k-1]`** and **`x[k]`** are out of order, thread T_k swaps them and sets a flag to indicate a swap has occurred.
        - Wait until all threads finish their work. This completes an even pass of this iteration.
    - **Odd Pass**:
        - Create at most $n/2$ threads: T_2, T_4, T_6, etc.
        - Thread T_k compares **`x[k-1]`** and **`x[k]`**. If **`x[k-1]`** and **`x[k]`** are out of order, thread T_k swaps them and sets a flag to indicate a swap has occurred.
        - Wait until all threads finish their work. This completes an odd pass of this iteration.
    - **Final Test**: Break this **`while`** loop if no swaps occurred because the array has been sorted.

In this concurrent version, if we ignore the cost of all thread creation and termination, we need at most $n/2$ even passes and $n/2$ odd passes (*i.e.*, $O(n)$ passes). Because in each even or odd pass, all comparisons are done concurrently, each pass only takes the time of comparing one pair of numbers (*i.e.*, $O(1)$ time). Thus, what we have done is a concurrent Even-Odd sort that requires $O(n)$ time or passes, each of which requires $n/2$ threads. This is n-fold faster than the sequential version! But, the total work is still the same (*i.e.*, $O(n^2)$ comparisons).

Program Logic

Write a program (*i.e.*, the **main**) to read in *n* integers into the array **x[*]**, and iterate at most *n* times. In each iteration, the **main** creates at most *n*/2 threads to execute an even pass and then another *n*/2 threads to execute an odd pass. Of course, the **main** must wait until all created threads of this pass complete before going for the next pass. Finally, if none of the even pass and odd pass swapped any numbers, the **main** prints out the sorted array.

Here are a few notes:

- The input array should be read in from **stdin**.
- The input array **x[*]** and the flag that indicated if any numbers have been swapped are global variables shared by all threads.
- You can only use the above program structure and the indicated thread creation and thread join. No other thread and/or process functions can be used for this program. Otherwise, you will receive a zero.

Input and Output

The input to your program should be taken from **stdin**. Your executable must be named as **prog3**. The command line looks like the following, where **input-filename** is a file from which **prog3** reads in the input values:

```
./prog3 < input-filename
```

The input file has the following format, where n is a positive integer, and x_0, x_1, ..., x_{n-1}, are n distinct integers. You may assume all input values being correct small numbers (*i.e.*, in the range of 0 and 999) so that you do not have to do error checking.

```
n
x0  x1  x2 ...  xn-1
```

Suppose the command line is

```
./prog3 < in.txt
```

and the file in.txt has the following lines:

```
8
7 1 3 2 8 4 5 9
```

Click here for a copy of this file.

Then, your program output should look like the following:

```
Concurrent Even-Odd Sort                          // from main()

Number of input data = 8                          // from main()
Input array:                                      // from main()
   7   1   3   2   8   4   5   9                  // from main()
                                                  // each number occupies 4 positions
                                                  // there has to be k = log2(n) runs
                                                  // from main(), do it for each run
Iteration i:                                      // from main(), run i
    ..........
    Even Pass:
        Thread k Created                          // from thread k
```

```
          ..........
            Thread k compares x[k-1] and x[k]          // from thread k
                                                       // thread k fills in the values of
                                                       //     x[k-1] and x[k]
            Thread k swaps x[k-1] and x[k]             // thread k swaps x[k-1] and x[k] if
                                                       //     they are out of order
                                                       // thread k fills in the values of
                                                       //     x[k-1] and x[k]
                                                       // otherwise, this message does not appear
          ..........
            Thread k exits                             // thread k exits
          ..........
         Odd Pass:
            Thread k Created                           // from thread k
          ..........
            Thread k compares x[k-1] and x[k]          // from thread k
                                                       // thread k fills in the values of
                                                       //     x[k-1] and x[k]
            Thread k swaps x[k-1] and x[k]             // thread k swaps x[k-1] and x[k] if
                                                       //     they are out of order
                                                       // thread k fills in the values of
                                                       //     x[k-1] and x[k]
                                                       // otherwise, this message does not appear
          ..........
            Thread k exits                             // thread k exits
          ..........
      Result after iteration i:                        // from main()
        aa bb cc dd ee ff gg hh                        // from main()
                                                       // use the input array format

      Final result after iteration j:                  // from main()
         1   2   3   4   5   7   8   9                 // from main()
                                                       // use the input array format
                                                       // note that the last iteration may not be n
                                                       // in fact, it may be lass than n
```

In the above sample output, the **`main()`** prints out the input and all intermediate result arrays. The **`main()`** iterates at most **`n`** times. All lines printed by the **`main()`** starts on column 1, and each data value is printed with 4 positions. All threads print their output lines with 8 leading spaces. **Make sure on each line you will print no more than 20 numbers.**

Submission Guidelines

General Rules

1. All programs must be written in C++.
2. Use **`Canvas`** to submit your work. This archive will contain both your program as well as your Makefile and README.
3. Unix filename is case sensitive, **`THREAD.cpp`**, **`Thread.CPP`**, **`thread.CPP`**, etc are ***not*** acceptable.
4. We will use the following approach to compile and test your programs:

   ```
   make               <-- make your program
   ./prog3 < in.txt   <-- test your program
   ```

   This procedure may be repeated a number of times with different input files to see if your program works correctly.
5. Your implementation should fulfill the program specifications as stated. Any deviation from the specification will cause you to receive **zero** point.

6. A **README** file is always required.
7. **No late submission will be graded.**
8. **Programs submitted to wrong class and/or wrong section will not be graded.**

Compiling and Running Your Programs

This is about the way of compiling and running your program. If we cannot compile your program due to syntax errors, wrong file names, etc, **we cannot test your program, and, as a result, you receive 0 point**. If your program compiles successfully but fails to run, **we cannot test your program, and, again, you receive 0 point**. **Therefore, before submitting your work, make sure your program can compile and run properly.**

1. **Not-compile programs receive 0 point.** By **not-compile**, I mean any reason that could cause an unsuccessful compilation, including missing files, incorrect filenames, syntax errors in your programs, and so on. Double check your files before you submit, since I will **not** change your program. Note again: Unix filenames are ***case sensitive***.
2. **Compile-but-not-run programs receive 0 point. Compile-but-not-run** usually means you have attempted to solve the problem to some degree but you failed to make it working properly.
3. **A meaningless or vague program receives 0 point even though it compiles successfully.** This usually means your program does not solve the problem but serves as a placeholder or template just making it to compile and run.

Program Style and Documentation

This section is about program style and documentation.

1. For each file, the first piece should be a program header to identify yourself like this:

```
// -----------------------------------------------------------
// NAME : John Smith                          User ID: xxxxxxxx
// DUE DATE : mm/dd/yyyy
// PROGRAM ASSIGNMENT #
// FILE NAME : xxxx.yyyy.zzzz (your unix file name)
// PROGRAM PURPOSE :
//    A couple of lines describing your program briefly
// -----------------------------------------------------------
```

   Here, **User ID** is the one you use to login. It is ***not*** your social security number nor your M number.

   For each function in your program, include a simple description like this:

```
// -----------------------------------------------------------
// FUNCTION  xxyyzz : (function name)
//     the purpose of this function
// PARAMETER USAGE :
//    a list of all parameters and their meaning
// FUNCTION CALLED :
//    a list of functions that are called by this one
// -----------------------------------------------------------
```

2. Your programs must contain enough concise and to-the-point comments. Do not write a novel!
3. Your program should have good indentation.
4. Do not use global variables!

Program Specifications

Your program must follow exactly the requirements of this programming assignment. Otherwise, you receive 0 point even though your program runs and produces correct output. The following is a list of potential problems.

1. Your program does not use the indicated algorithms/methods to solve this problem.
2. Your program does not follow the structure given in the specifications. For example, your program is not divided into functions and files, etc when the specification says you should.
3. Any other significant violation of the given program specifications.
4. **Incorrect output format.** This will cost you some points depending on how serious the violations are. The grader will make a decision. Hence, carefully check your program output against the required one.
5. **Your program does not achieve the goal of maximum parallelism.**

Program Correctness

If your program compiles and runs, we will check its correctness. We normally run your program with several sets of input data, one posted on this programming assignment page (the public one) and the others prepared by the grader (the private ones). You program must deliver correct results for all data sets. Depending on the seriousness of the problem(s), significant deduction may be applied. For example, if your program delivers all wrong results for the public data set, you receive 0 point for that component.

The `README` File

A file named **`README`** is required to answer the following questions:

1. **Question:** Are there any race conditions in this even-odd sort as suggested? Why?
2. **Question:** Prove rigorously that this algorithm does sort the input number correctly and takes no more than *n* iterations to sort an array of *n* numbers.
3. **Question:** In each iteration, the **`main()`** does the creation and join for the completion of **`n/2`** threads twice, once for an even pass and the other for an odd pass. Compared with simple comparisons, it requires a significant amount of time in creating and joining threads. If you are allowed to use extra variables/arrays and busy waiting, can you just create **`n`**/2 threads and let them do both the even pass and the odd pass in the same iteration without race conditions and still deliver correct results? More precisely, thread T_k compares **`x[k-1]`** and **`x[k]`** in an even pass, and then compare **`x[k]`** and **`x[k+1]`** in an odd pass? Suggest a solution and discuss its correctness.
4. **Question:** Furthermore, can you just create **`n`**/2 threads at the very beginning and let them do all the even pass and odd pass comparisons? In this way, you save more time on creating and joining threads. Suggest a solution and discuss its correctness.

You should elaborate your answer and provide details. **When answering the above questions, make sure each answer starts with a new line and have the question number (*e.g.*, Question X:) clearly shown. Separate two answers with a blank line.**

Note that the file name has to be **`README`** rather than **`readme`** or **`Readme`**. Note also that there is ***no*** filename extension, which means filename such as **`README.TXT`** is ***NOT*** acceptable.

`README` must be a plain text file. We do not accept files produced by any word processor. Moreover, watch for very long lines. More precisely, limit the length of each line to no more than 80 characters with the Return/Enter key for line separation. Missing this file, submitting non-text file, file with long lines, or providing incorrect and/or vague answers will cost you many points. Suggestion: Use a Unix text editor to prepare your **`README`** rather than a word processor.

Final Notes

1. Your submission should include the following files:
   - File **`thread.h`** that contains all class definitions of your threads.
   - File **`thread.cpp`** contains all class implementations of your threads.
   - File **`thread-main.cpp`** contains the main program.
   - File **`Makefile`** is a makefile that compiles the above three files to an executable file **`prog3`** without visualization. **Your makefile should make sure all paths are correct. Do not assume the grader knows your local path!**
   - The **`README`** file.

   Note also that without following this file structure your program is likely to fall into the compile-but-not-run category, and, as a result, you may get low grade. Therefore, before submission, check if you have the proper file structure and a correct makefile.
2. **Always start early, because I will not grant any extension if your home machine, network connection, your phone line or the department machines crash in the last minute.**
3. **Since the rules are all clearly stated, no leniency will be given and none of the above conditions is negotiable. So, if you have anything in doubt, please ask for clarification.**
4. **Click here to see how your program will be graded.**