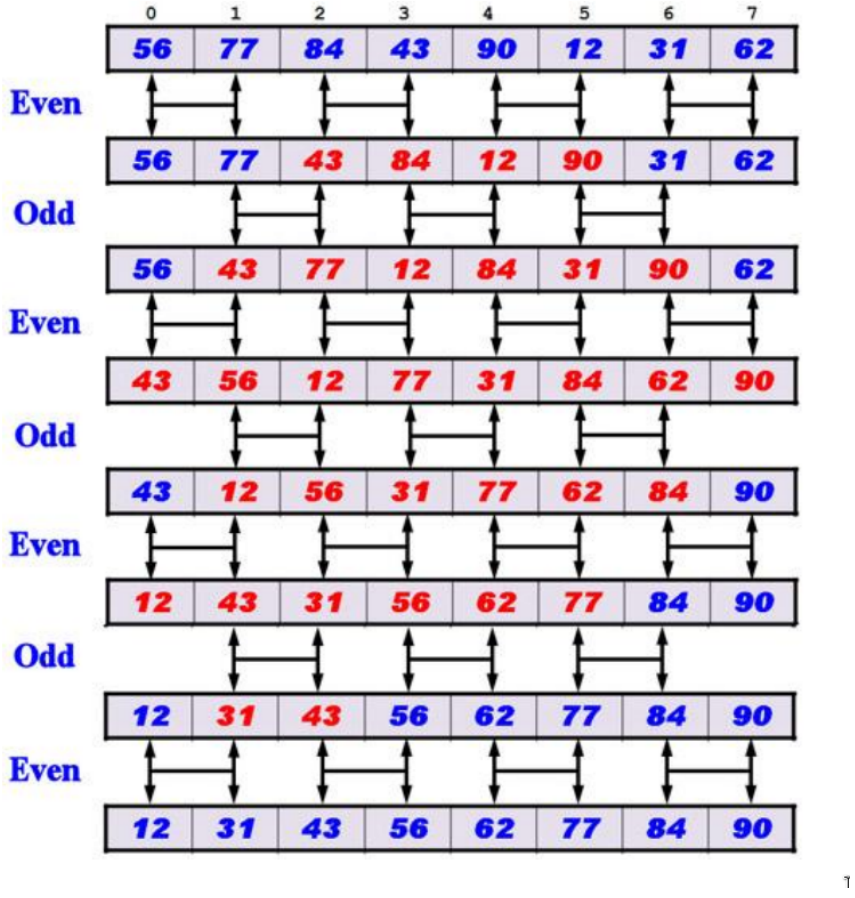
Programming Assignment III

CS3331

Spring 2025

1 The Even-Odd Sorting Algorithm

The even-odd sorting algorithm is a simple sorting algorithm that works by swapping out-of-order elements in alternating passes of even & odd pairs. Consider the sequence $(56, 77, 84, 43, 90, 12, 31, 32)$ as shown in the diagram below.



An Even pass means we compare two adjacent array elements such as x[2 i] and x[2 i +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[2 i +1] and x[2 i +2] (i.e., x[1] and x[2], x[3] and x[4], et cetera).

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 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 swaps were found in both passes we stop because the array has been sorted.

1.1 Sequential Algorithm

It is trivial to write a sequential program for 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;
  bool 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   */
void EvenOddSort(int x[], int n) {
  bool 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]`, et cetera. For an odd phase, it compares `x[2]` and `x[1]`, `x[4]` and `x[3]`, `x[6]` and `x[5]`, et cetera. Therefore, the initial indices are 1 and 2 for an even pass and odd pass respectively. The last index of both passes is at most $n - 1$.

The complexity of the even-odd sort algorithm 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.

1.2 Concurrent Algorithm

Even-odd sort can easily be converted to a concurrent algorithm. The concurrent version of the algorithm is given as pseudocode below. Note that each of the $n/2$ comparisons in a given pass are independent of

one another, and as such may 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 can simply kill all threads and create $n/2$ new threads to perform the next pass. Alternatively, if we are clever, we could use the original $n/2$ threads to sort a different pair of adjacent elements; however, you must be careful in order to prevent race conditions. As such, we will simply kill the old threads and create new ones in each loop.

1. Let the input to our algorithm be an array A.
2. In a loop, repeat the following up to n times.
   - Even Pass
     - Spawn $n/2$ threads $T_1, T_3, T_5, \ldots$
     - Thread T_k compares x[k−1] and x[k]. If they are out of order, the values in these locations are swapped.
     - Wait for all threads to finish their work.
   - Odd Pass
     - Spawn $n/2$ threads $T_2, T_4, T_6 \ldots$
     - Thread T_k compares x[k−1] and x[k]. If they are out of order, the values in these locations are swapped.
     - Wait for all threads to finish their work.
   - If no swap occurs during the even or odd pass, then break from the loop, as the array is now 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 in parallel, 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).

2 Requirements

You are tasked with writing a C++ program using the ThreadMentor library to implement the concurrent version of Even-Odd sorting described in section 1.2. In each iteration, the main thread 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 thread 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.

2.1 Input

Your program will read input from `stdin`. We will test your program using `stdin`-redirection, like so:

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

The input will have the following format:

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

where n is the number of elements in the input array x, and x_i is the ith element of said array. It is recommended that you use the `scanf()` standard library function to read the input.

An example input may be a file named `input1`:

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

which we may pass to our program using `stdin` redirection like so:

```
./prog3 < input1
```

2.2 Output

Below is an example output from the program.

```
Concurrent Even-Odd Sort

Number of input data = 8
Input array:
   7   1   3   2   8   4   5   9
Iteration 1:
    Even Pass:
     Thread 1 created
     Thread 1 compares x[0] = 7 and x[1] = 1
     Thread 1 swaps x[0] = 1 and x[1] = 7
     Thread 1 exits
     Thread 3 Created
     Thread 3 compares x[2] = 3 and x[3] = 2
     Thread 3 swaps x[2] = 2 and x[3] = 3
     Thread 7 Created
     Thread 3 exits
     Thread 7 compares x[6] = 5 and x[7] = 9
     Thread 7 exits
     Thread 5 Created
     Thread 5 compares x[4] = 8 and x[5] = 4
     Thread 5 swaps x[4] = 4 and x[5] = 8
     Thread 5 exits
    Odd Pass:
     Thread 2 Created
     Thread 2 compares x[1] = 7 and x[2] = 2
     Thread 6 Created
     Thread 2 swaps x[1] = 2 and x[2] = 7
     Thread 4 Created
     Thread 2 exits
     Thread 6 compares x[5] = 8 and x[6] = 5
     Thread 4 compares x[3] = 3 and x[4] = 4
     Thread 6 swaps x[5] = 5 and x[6] = 8
     Thread 4 exits
     Thread 6 exits
Result after iteration 1:
   1   2   7   3   4   5   8   9

 ... Repeat for up to n iterations ...
```

```
    Thread k exits
Result after iteration i:
   1   2   3   4   5   7   8   9
Final result after iteration i:
   1   2   3   4   5   7   8   9
```

Messages from the worker threads start on column 6 (that is, indented with 5 spaces). Indicators for the even & odd pass start on column 4. The rest of the print statements from the main process start on column 1.

Each worker thread will print up to 4 messages.

1. A creation message:

   `Thread k created`

2. A comparison message:

   `Thread k compares x[k-1] = v and x[k] = v`

3. A swap message, if the values are to be swapped:

   `Thread k swaps x[k-1] = v and x[k] = v`

4. An exit message:

   `Thread k exits`

3 Submission Guidelines

The following sections elaborate submission requirements not directly related to the semantics of the program itself.

3.1 General Rules

1. The program must be written in C++, programs written in other languages will receive a 0.
2. The program must use ThreadMentor, programs that make use of non-ThreadMentor concurrency/synchronization libraries will receive a 0.
3. Your work must be submitted as a `.zip` file via canvas. This `.zip` will contain your source code, your `Makefile`, and your `README`.
4. Your `Makefile`, when passed the `noVisual` target, should produce an executable file named `prog3` using the non-visual version of ThreadMentor.
5. We will use the following approach to compile and test your program:

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

   This procedure may be repeated a number of times with different input configurations to see if your program behaves correctly.
6. Your implementation should fulfill the program specification as stated.

3.2 Program Style and Documentation

- At the top of each `.cpp`/`.h` file, the first comment should be a program header to identify yourself like so:

```
// ------------------------------------------------------------
// NAME : John Smith                          User ID: xxxxxxxx
// DUE DATE : mm/dd/yyyy
// PROGRAM ASSIGNMENT #n
// FILE NAME : xxxx.yyy
// PROGRAM PURPOSE :
// A couple of lines describing your program briefly
// ------------------------------------------------------------
```

  Here, User ID is the one you use to log in to MTU systems.

- Your programs must contain a reasonable amount of concise and to-the-point comments. Do not write a novel.
- Your program should have reasonable indentation. Pyramids of whitespace are hard to parse!
- You should avoid using global variables. All of the monitor's state should be encapsulated in the monitor class itself.

3.3 The `README` File

A `README` file should be included with your submission. This file must either be a plaintext file named `README`, or a PDF file named `README.pdf`. The `README` should answer the questions enumerated below.

1. Are there any race conditions in the conccruent even-odd sort described in section 1.2?
2. In each iteration, the main ( ) function creates and joins the $n/2$ threads twice, once for an even pass and the other for an odd pass. While the simple comparisons these threads are performing are cheap, it requires a significant amount of time to create and join a thread. If you were allowed to use extra variables/arrays and busy waiting, would it be possible to just create $n/2$ threads and let them do both the even and 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 compares x [k] and x [k+1] in an odd pass? Suggest a solution and discuss its correctness.
3. Working from your solution to the prior question, could you just create $n/2$ threads at the very beginning and let them do all the even and odd pass comparisons? In this way, you save more time on creating and joining threads. Again, suggest a solution and discuss its correctness.

Please ensure you clearly indicate which question is being answered at a given point within your `README`. Make sure you have a convincing argument for each question!

3.4 Submission File Structure

Your submission should follow this file structure:

```
submission/
  thread-main.cpp   # main program implementation
  thread.h          # thread definitions
  thread.cpp        # thread implementations
  README{.pdf,.txt,}
  Makefile
```

Please do not write code in files outside the ones listed above, as you may lose points for not following the program file structure. Remember that file names in UNIX-like systems are case-sensitive, so `README.txt` is a different file from `readme.txt`!

4 Collaboration Policy

Collaboration is NOT allowed on this programming assignment. This includes the "Empty Hands" policy.

5 Final Remarks

While the concurrent even-odd sorting algorithm (and thus, by extension, the program you will write) is simple, it can be a little difficult to set up and get acquainted with the ThreadMentor library! As such, please familiarize yourself with the video on setting up and using ThreadMentor, as well as the documentation online at the URL:

`https://pages.mtu.edu/~shene/NSF-3/e-Book/index.html`

Get started on the program early, so if you run into issues with setting up ThreadMentor, you have time to contact the TA for assistance.

Finally, run through this quick checklist before and after you submit!

- My submission consists entirely of my own work, and I know I risk failing the entire course if I have plagiarized from any source.
- I am submitting the correct program to the correct assignment on canvas.
- My program follows the program file structure described in 3.4.
- I have included a `Makefile`.
- I have included my `README` file, and it is either a plaintext or PDF file.
- I have tested to make sure my `Makefile` works correctly, and successfully compiles the program using the command given in section 3.1.
- I have tested my program to ensure it works with several inputs, including the given example input.
- My program's output conforms to the output requirements described in section 2.2.
- My program takes input from `stdin`, as described in section 2.1.
- My program is written in C++.
- My program uses only ThreadMentor for concurrency and synchronization. I do not use pthreads or any other threading library.
- I have tested both compiling and running my program on a CS lab machine, or `colossus.it.mtu.edu`.
- My program runs the threads of a pass concurrently, I haven't accidentally sequentialized the computation!