Simple Sorting Simulation
Report

Hassan Hassan

21063424

Operating systems COMP2006

School of Electrical Engineering, Computing and Mathematical Sciences,
Curtin University

Introduction

This report outlines how I implemented a sorting simulation in C using two threads. The main goal of the assignment was to explore how threads can be used to solve a problem in parallel, with a focus on synchronization and safe access to shared data.

The sorting method used is called Odd-Even Transposition Sort, and I used POSIX threads (pthreads) to carry it out. Two threads, T1 and T2, take turns sorting pairs of numbers in an array. T1 looks at even-indexed pairs like A[0] & A[1], A[2] & A[3], etc., while T2 focuses on odd-indexed ones like A[1] & A[2], A[3] & A[4]. They keep alternating until the array is fully sorted.

The input comes from a file named ToSort, which contains up to 200 integers separated by spaces. After sorting, the program prints out the final sorted list, the total number of swaps made, and how many swaps were done by each thread.

Shared Data and Synchronisation

Since both threads are working with the same array and variables, I had to be careful to avoid race conditions. To keep things thread-safe, I used a mutex and two condition variables. These help make sure only one thread is modifying shared data at a time, and that the threads run in the correct order.

The shared array ***A*** holds the numbers we're sorting. Both threads read from and write to it, so locking is important. The variable n keeps track of how many numbers were read from the input file. Since it's only written once at the beginning, it doesn't need protection during the sort.

To track progress, I used several shared counters:

- swaps — total number of swaps done by both threads.
- t1_total_swaps and t2_total_swaps — how many swaps each thread has done overall.
- t1_swapped and t2_swapped — how many swaps were made in the most recent pass.

There's also a ***turn*** variable that helps the threads alternate. If ***turn*** is 0, it's T1's go; if it's 1, it's T2's. The ***done*** flag gets set when the array is sorted and no more swaps are needed. Once that happens, both threads exit cleanly.

When a thread finds that it's not their turn, it waits on its condition variable (condt1 or condt2). Once one thread finishes its turn, it signals the other to continue. This back-and-forth continues until the array is sorted

Correctness and Testing

To make sure the program worked properly, I tested it with different types of input. I started with a small array of 15 random numbers. Both threads alternated as expected, and the final result was sorted correctly. The number of swaps matched what I expected based on a manual run-through.

Next, I tried a larger test with 100 random integers. The program handled it without any problems. The threads continued to take turns, and the sorting finished as expected. The final sorted array was accurate, and both thread swap counts were reported correctly.

I also tested the worst-case scenario: a reverse-sorted array of 200 numbers, which is the maximum allowed by the program. This input led to the maximum number of swaps 19,900 in total and the program still ran correctly. It took a bit longer, but the threads alternated smoothly, and the final output was exactly right.

In all cases, I didn't run into any deadlocks or synchronization issues. The threads followed their alternating pattern, and the sorting completed when it should have.

Limitations and Known Issues

Although the program works well within the limits of the assignment, there are a few things that could be better.

First, Odd-Even Transposition Sort isn't the fastest algorithm out there. Its time complexity is $O(n^2)$ [1], which means it doesn't scale well. It works fine for up to 200 elements, but anything beyond that would get slow quickly.

Second, the program assumes that the input file is correctly formatted. If there are non-numeric characters or invalid input, it might not handle that gracefully. Error handling could definitely be improved in a future version.

Lastly, the way the threads alternate — using condition variables and blocking — introduces some waiting time. One thread is always idle while the other runs. It would be interesting to explore a more advanced implementation that reduces this idle time, maybe by overlapping some work.

Despite these issues, the program does what it's supposed to do. It sorts the array correctly, uses threads in a synchronized way, and meets the requirements of the assignment without crashing or hanging.

Source code

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

//synchronization tools
pthread_mutex_t mutex;
pthread_cond_t condt1, condt2;

int A[200];                    // array to be sorted
int n = 0;                     // number of elements in array
int swaps = 0;                 // number of swaps made
int t1_swapped = 1, t2_swapped = 1;    // tracks swaps in each round
int done = 0;                  // used to end threads
int turn = 0;                  //whose turn it is, 0 is t1 and 1 is t2
int t1_total_swaps = 0, t2_total_swaps = 0;  // track the total swaps done by each thread

// reads integers from file into array A
void read_input(char* filename) {
  FILE* fp = fopen(filename, "r");
  if(fp == NULL ) {
     perror("Error opening file");
     exit(1);
  }

  while(fscanf(fp, "%d", &A[n]) == 1 && n < 200) {
    n++;
  }

  fclose(fp);
}

// prints the array
void print_array() {
  for(int i = 0; i < n; i++) {
    printf("%d ", A[i]);
  }

  printf("\n");
}
```

```
// sorting thread function for t1 and t2
void* sort(void* arg) {

   char* type = (char*) arg;   // "T1" or "T2"

   int id = 0;


   if(type[1] == '1')
   {
      id = 0;    // id = 0 for T1
   } else {
      id = 1;    // id = 0 for T1
   }

   while(!done) {

      pthread_mutex_lock(&mutex);

      //wait for turn or arrays is sorted
      while(turn != id && !done) {
         if(id == 0)
         {
            pthread_cond_wait(&condt1, &mutex);    // t1 waits
         } else {
            pthread_cond_wait(&condt2, &mutex);    // t2 waits

         }
      }

      // exit if sorting is already done
      if(done) {
         pthread_mutex_unlock(&mutex);
         break;
      }

      int local_swaps = 0;

      // t1 sort
      if(id == 0) {
         for (int j = 0; j + 1 < n; j += 2) {
            if (A[j] > A[j + 1]) {
               int temp = A[j];
               A[j] = A[j + 1];
               A[j + 1] = temp;
```

```
            local_swaps++;
        }
      }
      t1_swapped = local_swaps;
      t1_total_swaps += local_swaps;


    // t2 sort
    } else {
      for(int j = 1; j + 1 < n; j += 2) {
        if (A[j] > A[j + 1]) {
          int temp = A[j];
          A[j] = A[j + 1];
          A[j + 1] = temp;
          local_swaps++;
        }
      }
      t2_swapped = local_swaps;
      t2_total_swaps += local_swaps;
    }

    swaps += local_swaps;



    // check if both threads made 0 swaps in round
    if(t1_swapped == 0 && t2_swapped == 0){
      done = 1;

      // signal both threads so they can exit if they're waiting
      pthread_cond_signal(&condt1);
      pthread_cond_signal(&condt2);
      pthread_mutex_unlock(&mutex);
      break;
    }

    // give turn to other thread
    turn = 1 - id;
    if(turn == 0) {
      pthread_cond_signal(&condt1);     // signal t1
    } else{
      pthread_cond_signal(&condt2);     // signal t2
    }


    pthread_mutex_unlock(&mutex);
  }
```

```
    pthread_t tid = pthread_self();

    if (id == 0) {
        printf("Thread ID %p (T1): total number of swaps = %d\n", tid, t1_total_swaps);

    } else {
        printf("Thread ID %p (T2): total number of swaps = %d\n", tid, t2_total_swaps);
    }


    return NULL;

}

int main(int argc, char* argv[]) {

    if(argc != 2){
        printf("Please provide the input file. Example: %s ToSort\n", argv[0]);
        return 1;
    }

    read_input(argv[1]);       // read array from file
    printf("Initial array %d numbers:\n", n);
    print_array();


    pthread_t t1, t2;

    // create threads
    pthread_create(&t1, NULL, sort, "T1");
    pthread_create(&t2, NULL, sort, "T2");

    // wait for threads to finish
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    printf("Sorted array:\n");
    print_array();

    printf("Total number of swaps to sort array A  = %d\n", swaps);

    //clean up
    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&condt1);
    pthread_cond_destroy(&condt2);
```

```
    return 0;
}
```

Sample Inputs and Outputs

To check that everything worked properly, I tested the program with a few different input cases. Below are three examples that show how the sorting behaves in different situations — a random list, an already sorted one, and a reverse-sorted one (worst case).

Sample input 1: Random 15 integers

Input:

5 3 11 2 1 4 5 1 10 11 21 17 25 16 6

Output:

Initial array 15 numbers:
5 3 11 2 1 4 5 1 10 11 21 17 25 16 6
Thread ID 0x16bdf3000 (T2): total number of swaps = 13
Thread ID 0x16bd67000 (T1): total number of swaps = 16
Sorted array:
1 1 2 3 4 5 5 6 10 11 11 16 17 21 25
Total number of swaps to sort array A = 29

The input list is unsorted. Both threads took turns comparing adjacent pairs and swapped as needed. After several passes, no more swaps were needed, and the array was fully sorted in ascending order

Sample input 2: Already Sorted (Best Case)

Input:

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

Output:

Initial array 20 numbers:
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
Thread ID 0x16da0f000 (T1): total number of swaps = 0
Thread ID 0x16da9b000 (T2): total number of swaps = 0

Sorted array:
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
Total number of swaps to sort array A = 0

Since the array is already sorted, both threads perform one round without making any swaps. The sorting condition is satisfied immediately and the program terminates correctly.

Sample input 3: Reverse Sorted (Worst Case)

Input:

199 198 197 ... 2 1 0 (200 values in descending order)

Output:

Thread ID 0x16d507000 (T2): total number of swaps = 9900
Thread ID 0x16d47b000 (T1): total number of swaps = 10000
Sorted array:
0 1 2 3 4 ... 198 199
Total number of swaps to sort array A = 19900

This was the worst-case input, requiring the maximum number of swaps. The program handled it without any issues. The threads alternated as expected, and the final result was sorted correctly.

References

[1]
Odd-Even Sort / Brick Sort
Link: https://www.geeksforgeeks.org/odd-even-sort-brick-sort/