Operating System Hw3

學號 1073007S
姓名 葉致廷

Part 1.

Explanation of hw3.c

Design flow:

Extract .txt into a 2D-interger-array

↓

Multithread Merge Sort

↓

Wirte the resulting array and the time spent into output.txt

Extract .txt into a 2D-interger-array:

1. Declare FILE pointer fp_in, fp_out to read test case and write the result into output.txt. Also, declare another FILE pointer fp_count to help us count the number of lines and the number of items in each line.

2. Declare `char *in_file, *out_file` to fetch command line arguments as `in_file = argv[1],` which is the pointer of the testcase file and `out_file = argv[2],` which is the pointer of the output file.

3. Open `in_file` with read mode using `fp_in = fopen(in_file, "r");` and catch the error if the file is not opened properly.

4. Then, count the number of lines and the number of items in each line using two helper function `int count(FILE * fp) and int* each_length(FILE * fp, int length)`

   - `int count(FILE * fp):` Count the numbers of line using `while(fgets(temp, LINE_MAX, fp))`. Here, `char temp[LINE_MAX] is declared as we pass it into fgets()`. In `while(fgets(temp, LINE_MAX, fp)),` we increment count_line by 1 until the condition no longer holds.

     - char * fgets (char * str, int num, FILE * stream);

     - `str`
       Pointer to an array of chars where the string read is copied.

     - `num`
       Maximum number of characters to be copied into *str* (including the terminating null-character).

     - `stream`
       Pointer to a FILE object that identifies an input stream.

- **int* each_length(FILE * fp, int length):** Count the numbers of items in each line, using int getc(FILE *stream) we can iterate through every character until the file is over. First, initialize each count_each_lines[i] to 1 as I found counting on '\n' might cause problems at the last line of test case. Next, whenever we encounter character, ' ', we increment count_each_lines[i] by 1, and also, when we encounter endl character, '\n', we increment i by 1 to count the next line.

- Since **int getc(FILE *stream)** move the address of fp, we redeclare fp_count = **fopen(in_file, "r")** to ensure the method starts from the head of the file.

5. **char **buffer = malloc(count_lines*sizeof(char*)):** A 2D-char array used as a buffer when we read in files later on. The allocated space of each row is set to LINE_MAX
   int **unsorted_array = malloc(count_lines*sizeof(int*)): A 2D-int array used as the source of the sorting procedure. The allocated space of each row is set to the element counts we received with **int* each_length(FILE * fp, int length)**

6. Use **void textfile_parser(char **buffer, int **unsorted_array, int *each_items_count, FILE *fp_in, int count)** to parse the textfile into 2D - int array.

   Use **while(fgets(buffer[i], LINE_MAX, fp_in))** to keep on reading the stream line by line until the stream ends and buffer[i] now stores each corresponding string.

   Inside the while loop**, buffer[i][strlen(buffer[i]) - 1] = '\0';** to replace '\n' with string ending '\0' to save one space.

   Then, use **char *token = strtok(buffer[i], " \n");** to get the first token from the string. **strtok(buffer[i], " \n")** splits the string with delimiter, " " and return. Also, using another while loop inside this scope**, while (token != NULL),** to keep fetching until we run out of tokens with the help of **token = strtok(NULL, " \n");**

   - **It gets the next token from the string, note that the use of NULL instead of the string in this case tells that it to carry on from where it left off.**

   and everytime we get a token we turn it into integer using **atoi(token)** and stores it into **unsorted_array[i][ipos]** and we increment ipos by 1 at the same time to proceed to index of the array. At the bottom of **while(fgets(buffer[i], LINE_MAX, fp_in))** we increment i by 1 to proceed to the next line of buffer.

   Note that if (ipos < count_each_lines[i] && !(isspace(*token))) is used to protect overflow from happening and also protect the input from trailing whitespaces.

   Soon as the outer for-loop ends, we finish our parsing process.

Multithread Merge Sort:

1. Before we start the multi thread procedure, we first implement the merge sort with the classical algorithm**.** Note that in **void merge(int *arr, int l, int m, int r); void mergesort(int *arr, int l, int r);** we add one more parameter **int *arr** to pass in the target array to improve the readibility of the code.

`void merge(int *arr, int l, int m, int r)`

Merges two subarrays of arr. First subarray is arr[l..m] Second subarray is arr[m+1..r]

Create two temp array to represent arr[l..m] and arr[m+1..r] and copy the data in arr[l..m] to L[], arr[m+1..r] to R[], then start comparing the items in L[] and R[]. Put the smaller item back into arr until one of L[] or R[] exsausted. Finally, Copy the remaining elements of L[] or R[], if there are any.

`void mergesort(int *arr, int l, int r)`

l is for left index and r is right index of the sub-array of arr to be sorted. If the starting index has not collide with the ending index, first find the middle point to divide the array into two halves then call mergesort for first half 3, next, call mergesort for second half, and finally, merge the two halves sorted in above.

2. Then, we implement three important components `typedef struct parameters; void *sortArray(void *params), void *mergeArray(void *params),` which will later be used in pthread_create.

```
typedef struct
{
    int from_index;
    int to_index;
    int *arr;

} parameters;
```

is used as a single argument that will be passed to start_routine, which contains the starting index of the sublist, the ending index of the sublist, and the list to be sorted.

`void *sortArray(void *params):`

1. Thread function to pass into the thread
2. `parameters* p = (parameters *)params; -` Get the data passed in by pthread_create
3. pass the parameters to mergesort to sort the sublist.
4. `pthread_exit(0);` Leave the child thread

`void *mergeArray(void *params):`

1. Thread function to pass into the thread
2. `parameters* p = (parameters *)params; -` Get the data passed in by pthread_create
3. Pass the part parameters to `void merge(int *arr, int l, int m, int r) to` merge the two sublist `.`
4. `pthread_exit(0);` Leave the child thread

`void Multithread_MergeSort(int *arr, int length):`

1. Declares three thread with `pthread_t threads[THREAD_COUNT]:` 2 for sorting and 1 for merging

2. Then, Establish the first sorting thread

   - Declare `parameters *data`
   - Set the stating index to `0`
   - Set the ending index to `0 + ((length-1)/2)`
   - Set the array passing in to `arr,` which is passed in as a parameter from the main procedure.
   - Create the first child thread with `pthread_create(&threads[0], 0, sortArray, data);` **`Note that the start_procedure we pass in is the sortArray function we defined earlier.`**

- Note that pthread_create creates a new thread and makes it executable. This routine can be called any number of times from anywhere within our code.
- **`pthread_create (pthread_t *thread, pthread_attr_t *attr, void *(*start_routine)(void *), void *arg)`**
  - **`argument:`**

  1. **thread:**

     An identifier for the new thread returned by the subroutine. This is a pointer to pthread_t structure. When a thread is created, an identifier is written to the memory location to which this variable points. This identifier enables us to refer to the thread.

  2. **attr:**

     An attribute object that may be used to set thread attributes. We can specify a thread attributes object, or **NULL for the default values.**

  3. **start_routine:**

     The routine that the thread will execute once it is created. We should pass the address of a function taking a pointer to void as a parameter and the function will return a pointer to void. So, we can pass any type of single argument and return a pointer to any type. Using a new thread explicitly provides a pointer to a function where the new thread should start executing.

4. **arg:**

A single argument that may be passed to start_routine. It must be passed as a void pointer. NULL may be used if no argument is to be passed.

3. Establish the second sorting thread

- Declare `parameters *data`
- Set the stating index to `((length-1)/2)+1`
- Set the ending index to `length-1`
- Set the array passing in to `arr,` which is passed in from the main procedure.
- Create the second child thread with

```
pthread_create(&threads[1], 0, sortArray, data);
```

Note that the start_procedure we pass in is the sortArray function we defined earlier.

4. Wait for the two sorting threads to finish using pthread_join.

- int pthread_join (pthread_t th, void **thread_return)
- argument:
- pthread_t th is the thread for which to wait, the identified that pthread_create filled in for us.
- void **thread_return is a pointer to a pointer that itself points to the return value from the thread. This function returns zero for success and an error code on failure.

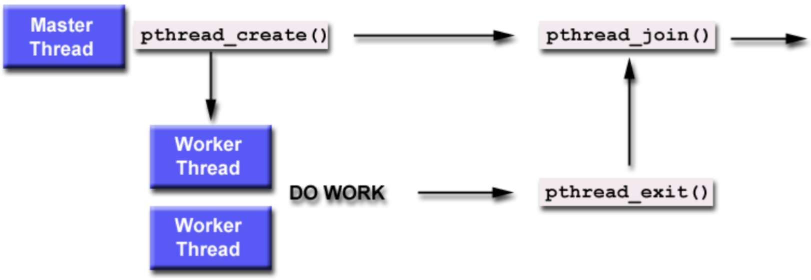
Here, we use

```
for (int i = 0; i < THREAD_COUNT - 1; i++)

      pthread_join(threads[i], NULL);
```

as threads[0] and thread[1] are the threads that we’re waiting,

and as they do not need to return anything, the second parameter is NULL.

The create - exit - join relation is depicted as follow



5. Finally, we establish the merge thread
    - Declare parameters *data
    - Set the stating index to 0
    - Set the ending index to length-1
    - Set the array passing in to arr, which is passed in from the main procedure.
    - Create the merging thread with
      `pthread_create(&threads[1], 0, mergeArray, data);`
      `Note that the start_procedure we pass in is the mergeArray function we defined earier.`

6. Wait for the merge thread to finish
    - `pthread_join(threads[2], NULL);`
      as threads[2] are the threads that we're waiting,
      and as they do not need to return anything, the second parameter is NULL.

Wirte the resulting array and the time spent into output.txt:

`for(int i = 0; i < count_lines; i++),` this file loop iterate over each line
In the for loop, `use clock_t begin = clock(); clock_t end = clock(); time_spent += (double)(end - begin)/CLOCKS_PER_SEC;` to get the duration time of each `Multithread_MergeSort(unsorted_array[i], count_each_lines[i]).`

Note that the parameter is used as the target array and the length, the length is used to calculate the starting and ending indexes in the merge sort.

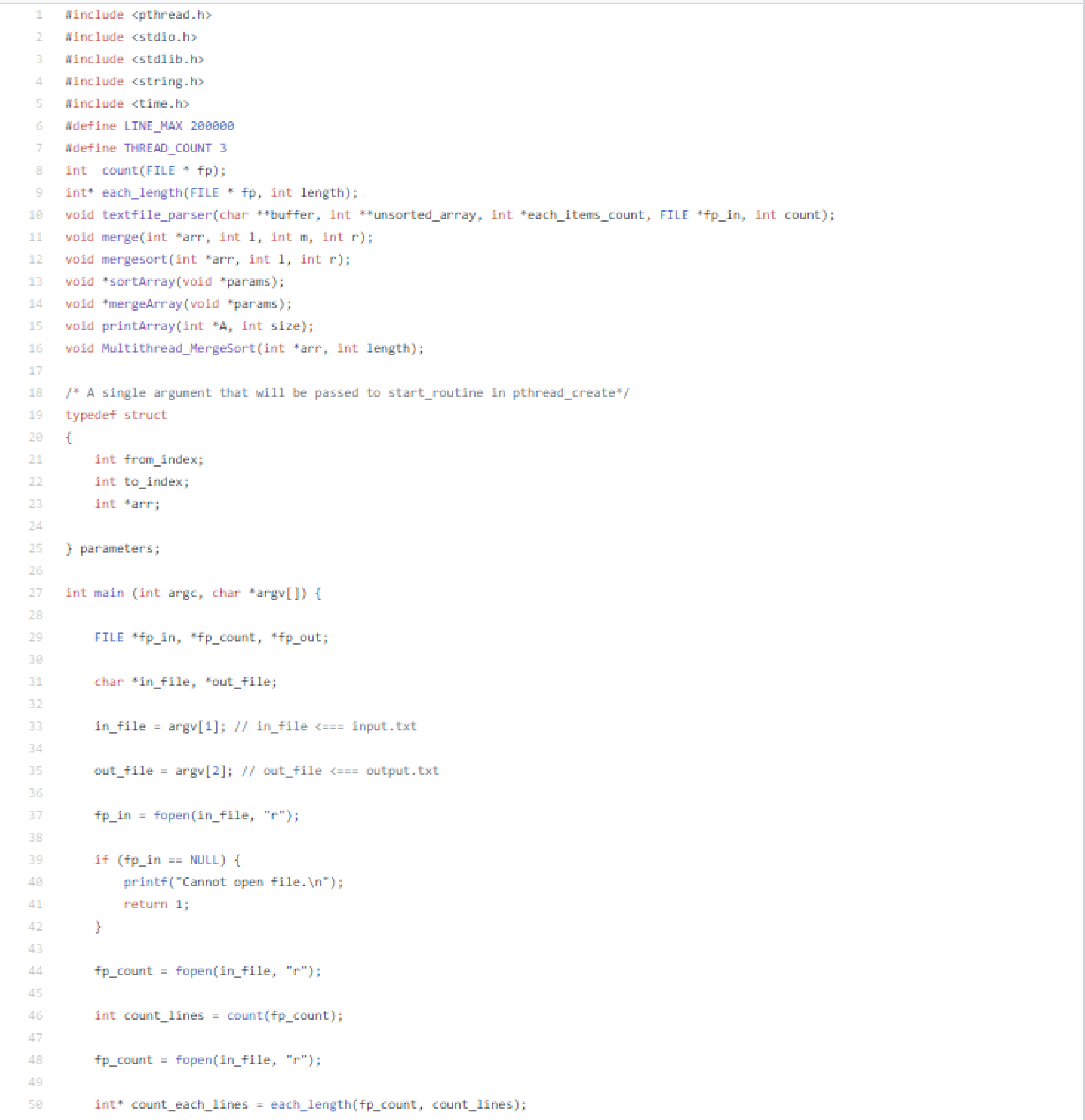
At the bottom of each iteration use a for loop to write the result in output.txt with `fprintf(fp_out, "%d ", unsorted_array[i][j]);` and a single line to write in the duration time with `fprintf(fp_out, "\nduration: %f\n\n", time_spent);`

After the write in procedure finish, close all the file stream with

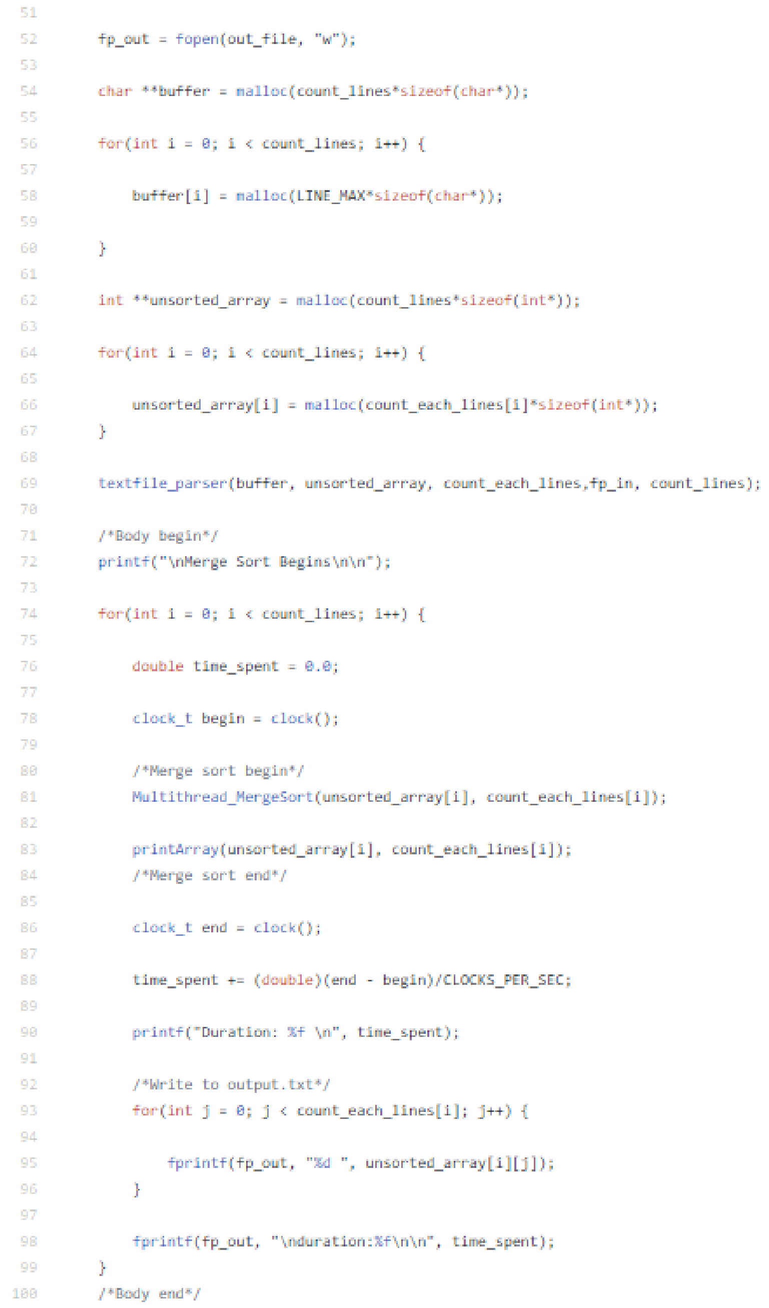
- fclose(fp_in);
- fclose(fp_count);
- fclose(fp_out);

Part 2.

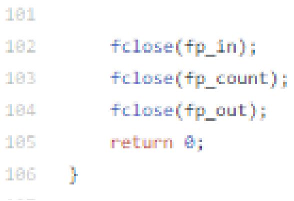
Code Screenshot



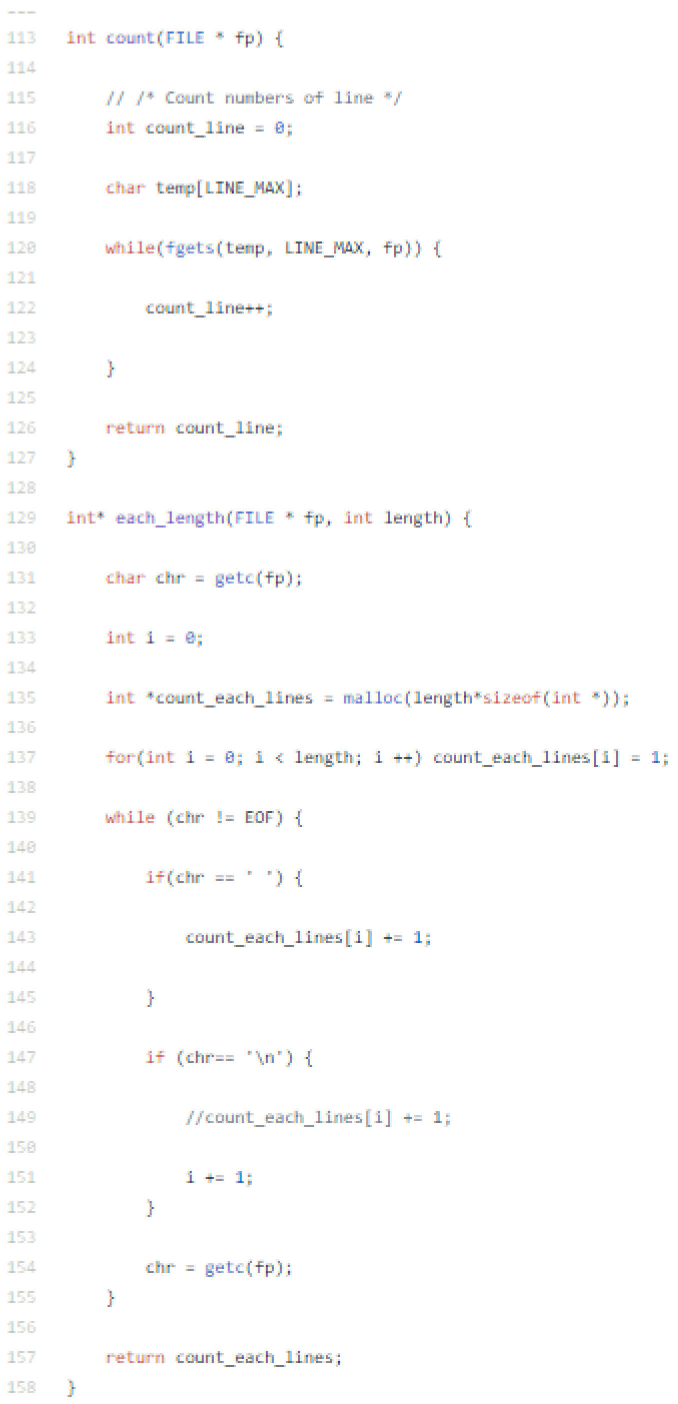
```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#define LINE_MAX 200000
#define THREAD_COUNT 3
int  count(FILE * fp);
int* each_length(FILE * fp, int length);
void textfile_parser(char **buffer, int **unsorted_array, int *each_items_count, FILE *fp_in, int count);
void merge(int *arr, int l, int m, int r);
void mergesort(int *arr, int l, int r);
void *sortArray(void *params);
void *mergeArray(void *params);
void printArray(int *A, int size);
void Multithread_MergeSort(int *arr, int length);

/* A single argument that will be passed to start_routine in pthread_create*/
typedef struct
{
    int from_index;
    int to_index;
    int *arr;

} parameters;

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

    FILE *fp_in, *fp_count, *fp_out;

    char *in_file, *out_file;

    in_file = argv[1]; // in_file <=== input.txt

    out_file = argv[2]; // out_file <=== output.txt

    fp_in = fopen(in_file, "r");

    if (fp_in == NULL) {
        printf("Cannot open file.\n");
        return 1;
    }

    fp_count = fopen(in_file, "r");

    int count_lines = count(fp_count);

    fp_count = fopen(in_file, "r");

    int* count_each_lines = each_length(fp_count, count_lines);
```



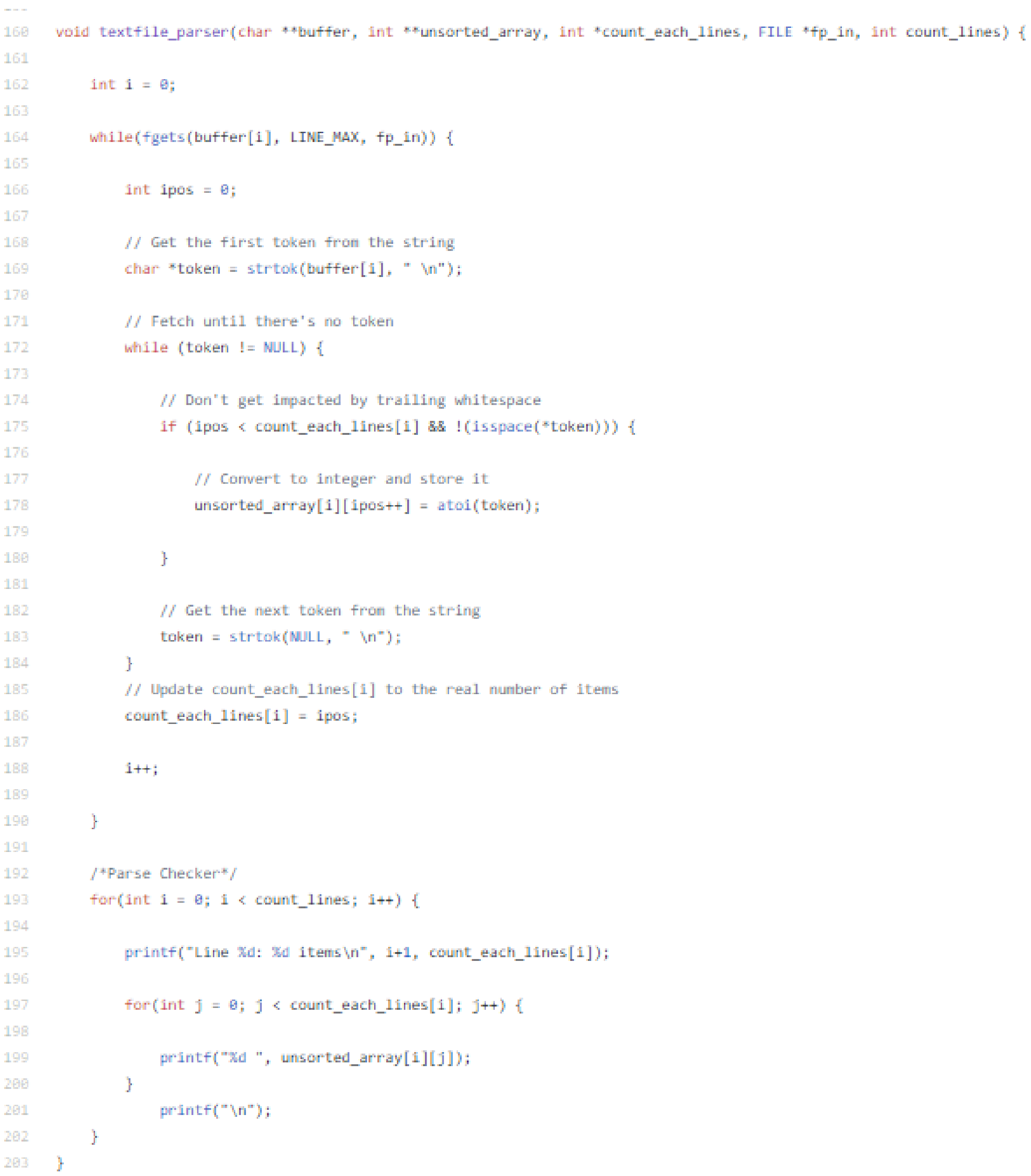
```
    fp_out = fopen(out_file, "w");

    char **buffer = malloc(count_lines*sizeof(char*));

    for(int i = 0; i < count_lines; i++) {

        buffer[i] = malloc(LINE_MAX*sizeof(char*));

    }

    int **unsorted_array = malloc(count_lines*sizeof(int*));

    for(int i = 0; i < count_lines; i++) {

        unsorted_array[i] = malloc(count_each_lines[i]*sizeof(int*));
    }

    textfile_parser(buffer, unsorted_array, count_each_lines,fp_in, count_lines);

    /*Body begin*/
    printf("\nMerge Sort Begins\n\n");

    for(int i = 0; i < count_lines; i++) {

        double time_spent = 0.0;

        clock_t begin = clock();

        /*Merge sort begin*/
        Multithread_MergeSort(unsorted_array[i], count_each_lines[i]);

        printArray(unsorted_array[i], count_each_lines[i]);
        /*Merge sort end*/

        clock_t end = clock();

        time_spent += (double)(end - begin)/CLOCKS_PER_SEC;

        printf("Duration: %f \n", time_spent);

        /*Write to output.txt*/
        for(int j = 0; j < count_each_lines[i]; j++) {

            fprintf(fp_out, "%d ", unsorted_array[i][j]);
        }

        fprintf(fp_out, "\nduration:%f\n\n", time_spent);
    }
    /*Body end*/
```



```

    fclose(fp_in);
    fclose(fp_count);
    fclose(fp_out);
    return 0;
}
```

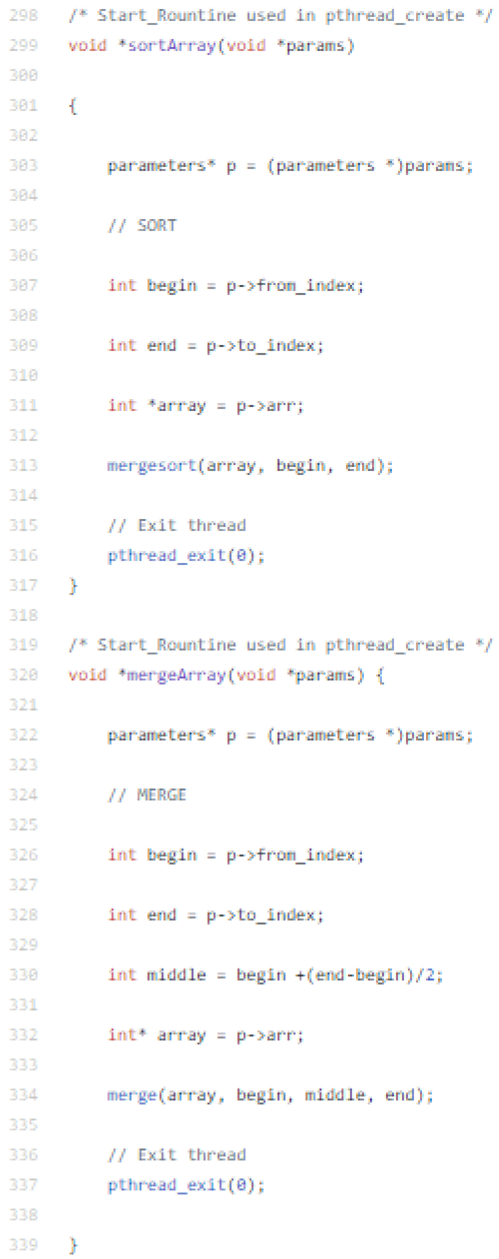


```
int count(FILE * fp) {

    // /* Count numbers of line */
    int count_line = 0;

    char temp[LINE_MAX];

    while(fgets(temp, LINE_MAX, fp)) {

        count_line++;

    }

    return count_line;
}

int* each_length(FILE * fp, int length) {

    char chr = getc(fp);

    int i = 0;

    int *count_each_lines = malloc(length*sizeof(int *));

    for(int i = 0; i < length; i ++) count_each_lines[i] = 1;

    while (chr != EOF) {

        if(chr == ' ') {

            count_each_lines[i] += 1;

        }

        if (chr== '\n') {

            //count_each_lines[i] += 1;

            i += 1;
        }

        chr = getc(fp);
    }

    return count_each_lines;
}
```

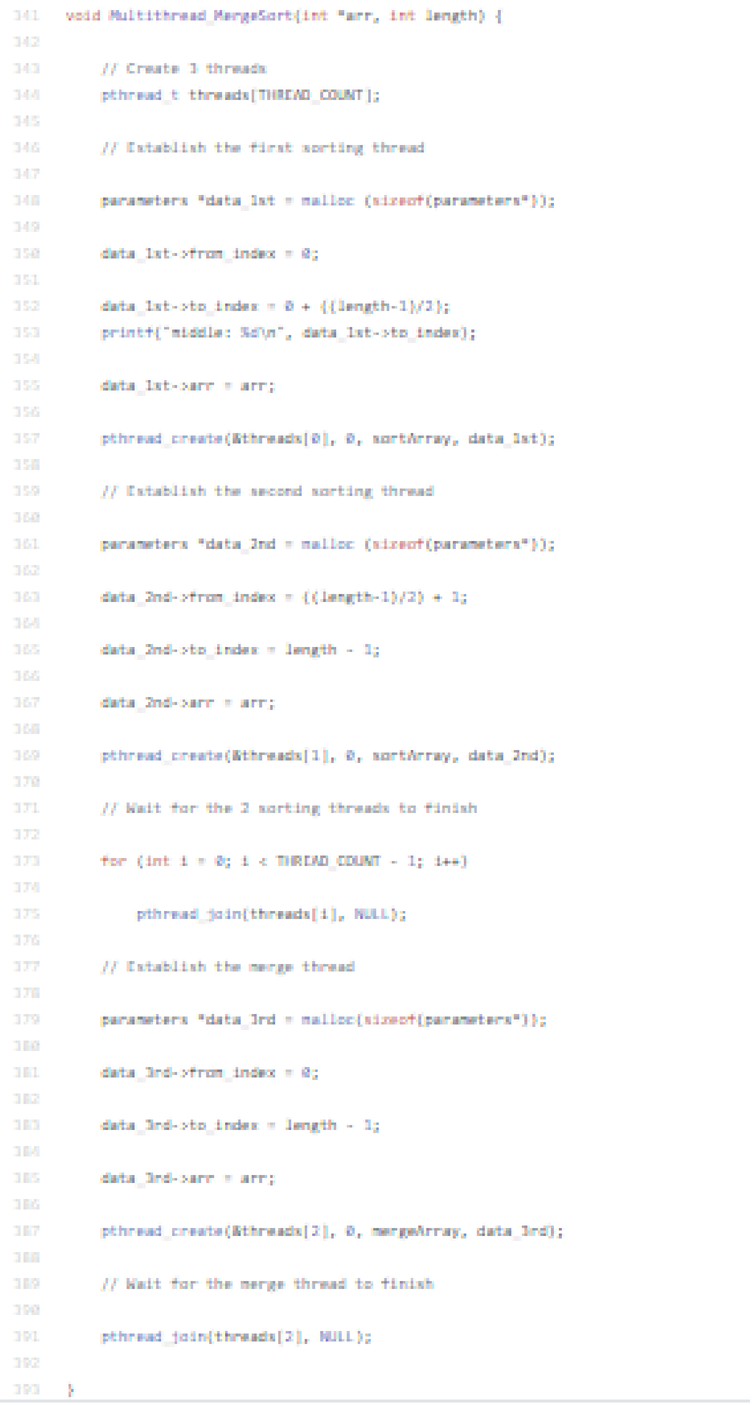


```
void textfile_parser(char **buffer, int **unsorted_array, int *count_each_lines, FILE *fp_in, int count_lines) {

    int i = 0;

    while(fgets(buffer[i], LINE_MAX, fp_in)) {

        int ipos = 0;

        // Get the first token from the string
        char *token = strtok(buffer[i], " \n");

        // Fetch until there's no token
        while (token != NULL) {

            // Don't get impacted by trailing whitespace
            if (ipos < count_each_lines[i] && !(isspace(*token))) {

                // Convert to integer and store it
                unsorted_array[i][ipos++] = atoi(token);

            }

            // Get the next token from the string
            token = strtok(NULL, " \n");
        }
        // Update count_each_lines[i] to the real number of items
        count_each_lines[i] = ipos;

        i++;

    }

    /*Parse Checker*/
    for(int i = 0; i < count_lines; i++) {

        printf("Line %d: %d items\n", i+1, count_each_lines[i]);

        for(int j = 0; j < count_each_lines[i]; j++) {

            printf("%d ", unsorted_array[i][j]);
        }
            printf("\n");
    }
}
```

```c

void merge(int *arr, int l, int m, int r)
{
    int i, j, k;

    int n1 = m - l + 1;

    int n2 =  r - m;

    // Create temp arrays
    int L[n1], R[n2];

    // Copy data to temp arrays L[] and R[]
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];

    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1+ j];

    /* Merge the temp arrays back into arr[l..r]*/

    i = 0; // Initial index of first subarray

    j = 0; // Initial index of second subarray

    k = l; // Initial index of merged subarray

    while (i < n1 && j < n2) {

        if (L[i] <= R[j]) {

            arr[k] = L[i];

            i++;

        }

        else {

            arr[k] = R[j];

            j++;
        }

        k++;
    }

    /* Copy the remaining elements of L[], if there
       are any */
    while (i < n1) {

        arr[k] = L[i];

        i++;

        k++;
    }

    /* Copy the remaining elements of R[], if there
       are any */
    while (j < n2) {

        arr[k] = R[j];

        j++;

        k++;
    }
}

void mergesort(int *arr, int l, int r)
{
    if (l < r) {

        int m = l+(r-l)/2;

        // Sort two halves
        mergesort(arr, l, m);
        mergesort(arr, m+1, r);
        // Merge two halves
        merge(arr, l, m, r);
    }
}

void printArray(int *arr, int size)
{

    for (int i=0; i < size; i++)
        printf("%d ", arr[i]);

    printf("\n");
}

```



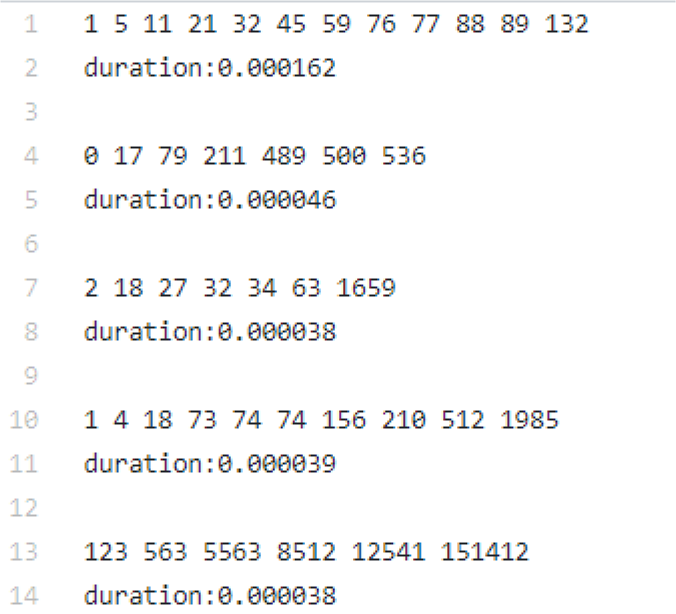
```
/* Start_Rountine used in pthread_create */
void *sortArray(void *params)

{

    parameters* p = (parameters *)params;

    // SORT

    int begin = p->from_index;

    int end = p->to_index;

    int *array = p->arr;

    mergesort(array, begin, end);

    // Exit thread
    pthread_exit(0);
}

/* Start_Rountine used in pthread_create */
void *mergeArray(void *params) {

    parameters* p = (parameters *)params;

    // MERGE

    int begin = p->from_index;

    int end = p->to_index;

    int middle = begin +(end-begin)/2;

    int* array = p->arr;

    merge(array, begin, middle, end);

    // Exit thread
    pthread_exit(0);

}
```



```c
void Multithread_MergeSort(int *arr, int length) {

    // Create 3 threads
    pthread_t threads[THREAD_COUNT];

    // Establish the first sorting thread

    parameters *data_1st = malloc (sizeof(parameters*));

    data_1st->from_index = 0;

    data_1st->to_index = 0 + ((length-1)/2);
    printf("middle: %d\n", data_1st->to_index);

    data_1st->arr = arr;

    pthread_create(&threads[0], 0, sortArray, data_1st);

    // Establish the second sorting thread

    parameters *data_2nd = malloc (sizeof(parameters*));

    data_2nd->from_index = ((length-1)/2) + 1;

    data_2nd->to_index = length - 1;

    data_2nd->arr = arr;

    pthread_create(&threads[1], 0, sortArray, data_2nd);

    // Wait for the 2 sorting threads to finish

    for (int i = 0; i < THREAD_COUNT - 1; i++)

        pthread_join(threads[i], NULL);

    // Establish the merge thread

    parameters *data_3rd = malloc(sizeof(parameters*));

    data_3rd->from_index = 0;

    data_3rd->to_index = length - 1;

    data_3rd->arr = arr;

    pthread_create(&threads[2], 0, mergeArray, data_3rd);

    // Wait for the merge thread to finish

    pthread_join(threads[2], NULL);

}
```

Part 3.

Output Screenshot



```
1 5 11 21 32 45 59 76 77 88 89 132
duration:0.000162

0 17 79 211 489 500 536
duration:0.000046

2 18 27 32 34 63 1659
duration:0.000038

1 4 18 73 74 74 156 210 512 1985
duration:0.000039

123 563 5563 8512 12541 151412
duration:0.000038
```