CMPE 180C Operating System Project: Multithreaded Sorting Application

Vilas U Dhuri
vilas.dhuri@sjsu.edu
MS, Computer Engineering
San Jose State University
San Jose, USA

Rishabh Gupta
rishabh.gupta02@sjsu.edu
MS, Computer Engineering
San Jose State University
San Jose, USA

Saharsh Shivhare
saharshanurag.shivhare@sjsu.edu
MS, Computer Engineering
San Jose State University
San Jose, USA

***Abstract*—This report is for CMPE 180C project for developing a multithreaded sorting program.**

Keywords—sort, merge sort, quick sort, selection sort, insertion sort, threads, operation systems

I. INTRODUCTION

Multithreading is the ability of a program or an operating system process to handle numerous requests from the same user at the same time without the need for several copies of the programming to be executing in the computer. Multithreading allows the application to divide its task into individual threads. In multi-threads, the same process or task can be done by the number of threads, or we can say that there is more than one thread to perform the task in multithreading. In operating system user-level thread and the kernel-level thread are the two types of threads. User-level threads were controlled independently of the kernel and hence without kernel support. The kernel-level threads, on the other hand, are managed directly by the operating system. [1]
The three established multithreading models are:

- Many to one multithreading model
- One to one multithreading model
- Many to Many multithreading models

Sorting algorithms are ways to reorganize a huge number of elements into a given order, for as from highest to lowest, or vice versa, or even alphabetically. These algorithms take an input list, process it (that is, perform operations on it), and then return a sorted list.[3] The sorting algorithms that we have implemented for this project are as follows:

- Merge Sort
- Quick Sort
- Insertion Sort
- Selection sort

Algorithms	***Best Case***	***Worst Case***
Merge Sort	Ω(N*logN)	O(N*logN)
Quick Sort	Ω(N*logN)	$O(N^2)$
Insertion Sort	Ω(N)	$O(N^2)$
Selection Sort	$\Omega(N^2)$	$O(N^2)$

Table 1: Time Complexities of sorting algorithms

II. PROBLEM STATEMENT

The aim of the project to write and develop a multithreaded sorting program. Given a list of integers, this list is divided into two smaller lists of equal size. Each sub-list is sorted by two distinct threads (labelled sorting threads) using the sorting method of your choice. A third thread—a merging thread—then merges the two sub-list into a single sorted list. The project should utilize sing Java's fork-join parallelism API. This project will be developed in two different versions. Each version will implement a different divide-and-conquer sorting *viz.* Quick Sort and Merge Sort. A graphical representation of the program is shown.

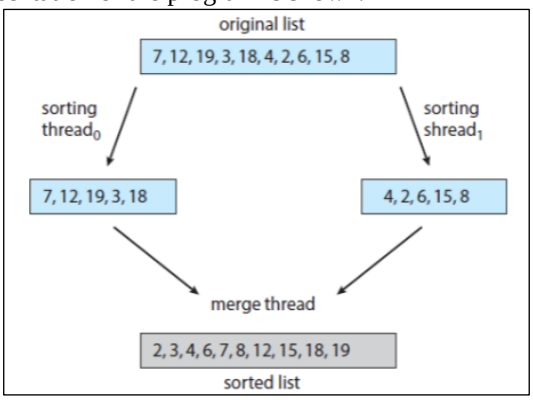

Figure 1

III. MULTITHREADING MODELS

A. Many to one multithreading model: Many user level threads are mapped to one kernel thread in the many to one architecture. This type of relationship makes it easier to create an effective context-switching environment, even on a simple kernel with no thread support. The downside of this architecture is that it cannot take advantage of the hardware acceleration offered by multithreaded processes or multi-processor systems because there is only one kernel-level thread schedule at any given moment. All thread management is done in user space in this case. This model blocks the entire system if blocking occurs.

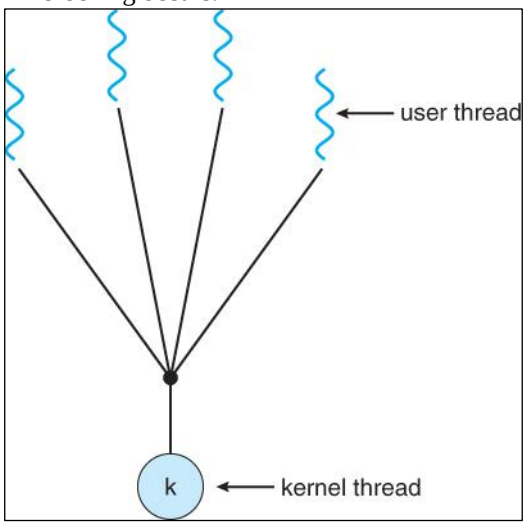

Figure 2

B. One to one multithreading model: A single user-level thread is mapped to a single kernel-level thread in the one-to-one architecture. Multiple threads can be run in parallel using this form of relationship. This advantage, however, has a disadvantage. Every new user thread must be accompanied by the creation of a kernel thread, resulting in overhead that might degrade the parent process' performance. The Windows and Linux operating systems aim to solve this problem by limiting the number of threads that can be created. [1][2]

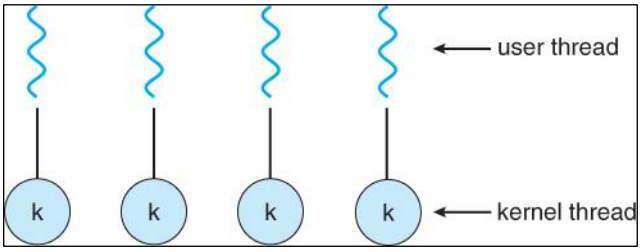

Figure 3

C. Many to Many multithreading models: There are numerous user-level threads and several kernel-level threads in this approach. The number of kernel threads generated is determined by the application. At both levels, the developer can build as many threads as he wants, but they may not all be the same. The many to many model is a hybrid of the two previous concepts. If a thread makes a blocked system call in this model, the kernel can schedule another thread to execute. In addition, the introduction of many threads eliminates the complexity that existed in previous models. Although this paradigm allows for the development of many kernel threads, it cannot accomplish full concurrency. This is due to the kernel's ability to only schedule one process at a time. [1][2]

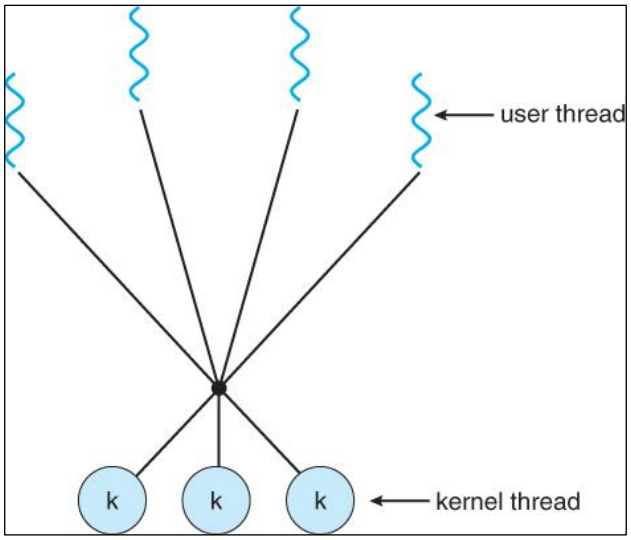

Figure 4

IV. SORTING ALGORITHMS

A. Quick Sort: The name "Quick Sort" stems from the fact that it can sort a list of data elements substantially faster (two or three times faster) than any other sorting algorithm. It is based on separating an array (partition) into smaller ones and swapping (exchange) based on a comparison with the 'pivot' element picked. Quick sort is sometimes known as "Partition Exchange" sort because of this. It selects a pivot element and partitions the specified array around that pivot. There are numerous versions of quickSort that select pivot in various ways. [3][4]

- Always pick first element as pivot.
- Always pick last element as pivot (implemented below)
- Pick a random element as pivot.
- Pick median as pivot.

Partitioning is the most important operation in quickSort ().A psuedo code for recursive quick sort is as follows:

```
/* low  → Starting index, high  → Ending index */
    quicksort(arr [ ], low, high)
    {
            if (low < high)
            {
            /* pi is partitioning index, arr[pi] is now
            at right place */
                    pi = partition (arr, low, high);
                    quicksort (arr, low, pi - 1);
                    quicksort (arr, pi + 1, high);
            }
    }
```

The flow of Quick Sort Algorithm is as follows:

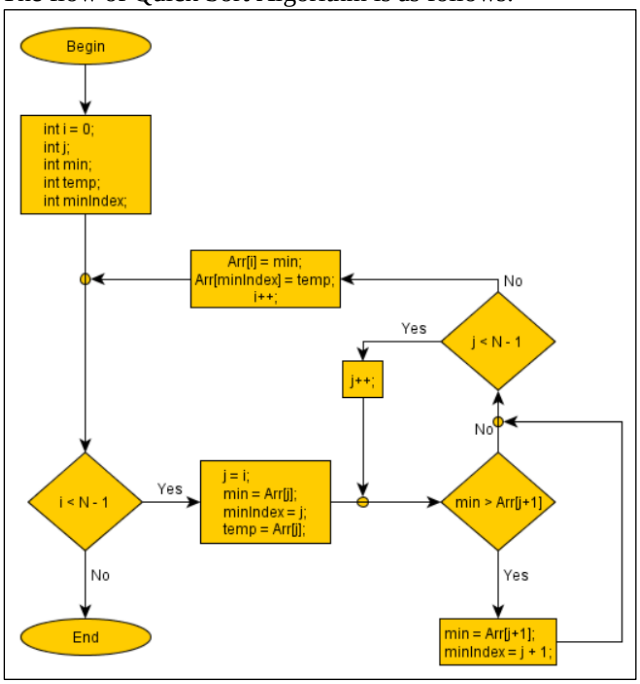

Figure 5

B. *Merge Sort:* One of the most efficient sorting algorithms is merge sort. It operates on the divide-and-conquer premise. Merge sort continuously cuts down a list into numerous sublists until each sublist contains only one entry, then merges those sublists into a sorted list. The merge() function joins two halves together. merge(arr, l, m, r) is a key process that assumes arr[l..m] and arr[m+1..r] are both sorted sub-arrays and merges them into one. Here, *l* is the start of array *arr*. *r* is the last position of the array and *m* is the middle postion. The flow of Merge Sort algorithm gicen below. The psuedo implementation is shown below.[3][4]

MergeSort(arr [ ], l, r)
If r > l

1. Find the middle point to divide the array into two halves:
   middle m = l+ (r-l)/2
2. Call mergeSort for first half:
   Call mergeSort(arr, l, m)
3. Call mergeSort for second half:
   Call mergeSort(arr, m+1, r)
4. Merge the two halves sorted in step 2 and 3:
   Call merge (arr, l, m, r)

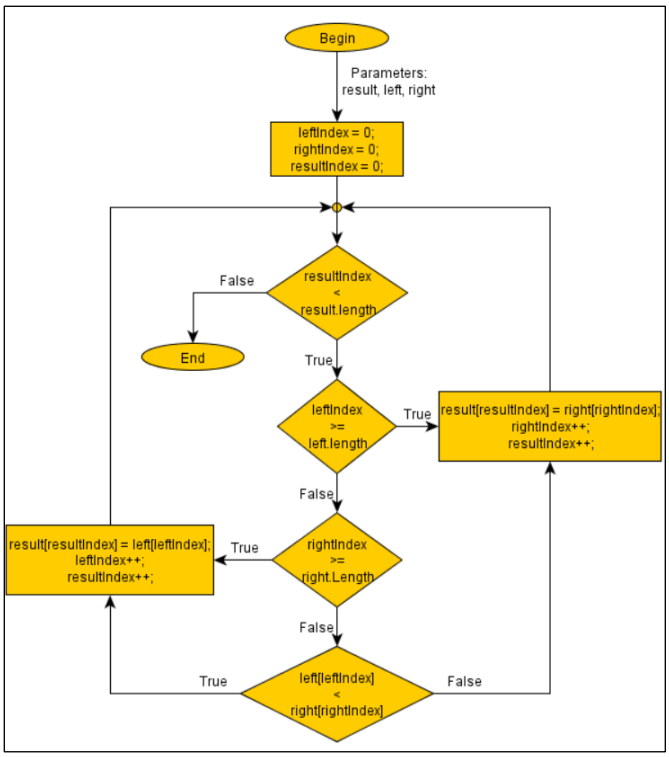

Figure 6

C. *Insertion Sort*: Insertion sort is a sorting mechanism that builds a sorted array one item at a time. The array members are compared progressively before being placed in a specific order. The analogy can be seen in the way we lay out a deck of cards. The name Insertion Sort comes from the notion of inserting an element into a specific location. The flow of insertion sort algorithm is as shown.

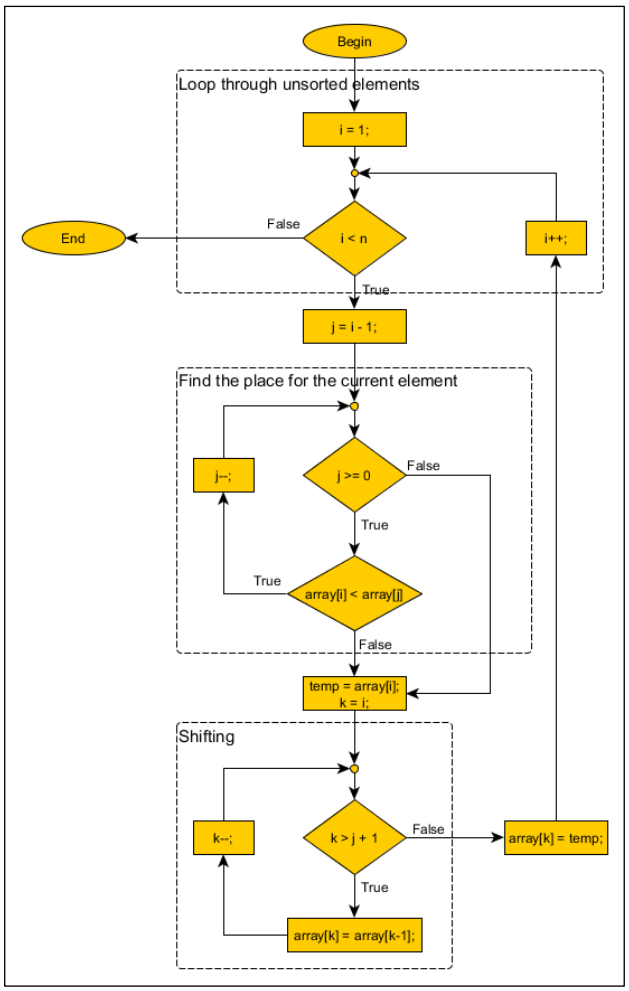

Figure 7

D. *Selection Sort:* A simple comparison-based sorting algorithm is selection sort. It is already installed and does not require additional memory. This algorithm's concept is straightforward. We divide the array into two parts: sorted and unsorted. The left subarray is sorted, whereas the right subarray is unsorted. The sorted subarray is initially empty, but the unsorted array contains the entire provided array. The flow of insertion sort algorithm is as shown.

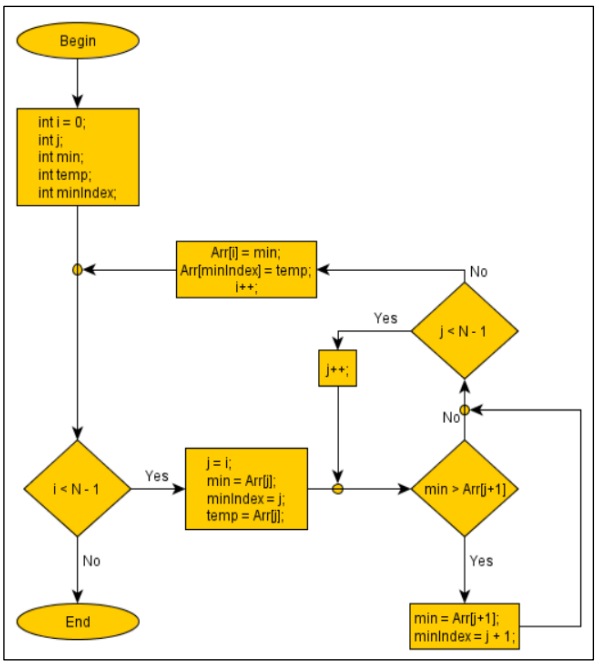

Figure 8

V. SOFTWARE ARCHITECTURE

For our Multithreaded Sorting Application, we ask the user to enter the number of integer elements he wants to sort. After the number is enter we display the users the number generated by the "rand( )" function. If the number of elements is greater than our selected threshold (i.e., 20) we ask the user whether he wants to perform "Quick sort" or "Merge Sort". Upon selection we feed the array of elements into two separate threads. One of these threads will sort the first half of the array and the other thread will sort the remaining items of the array. The sorting algorithm will be used as per the user desired algorithm. Since we are using recursive method to sort, we keep on decreasing the array size in recursive functions. Once the number of elements is less than the threshold, "Insertion Sort" is performed. On completion of the sorting, the two threads are merged and finally we display the sorted array and the time it took sort the array.

Following is the flow of our implemented Multithreaded Sorting Application:

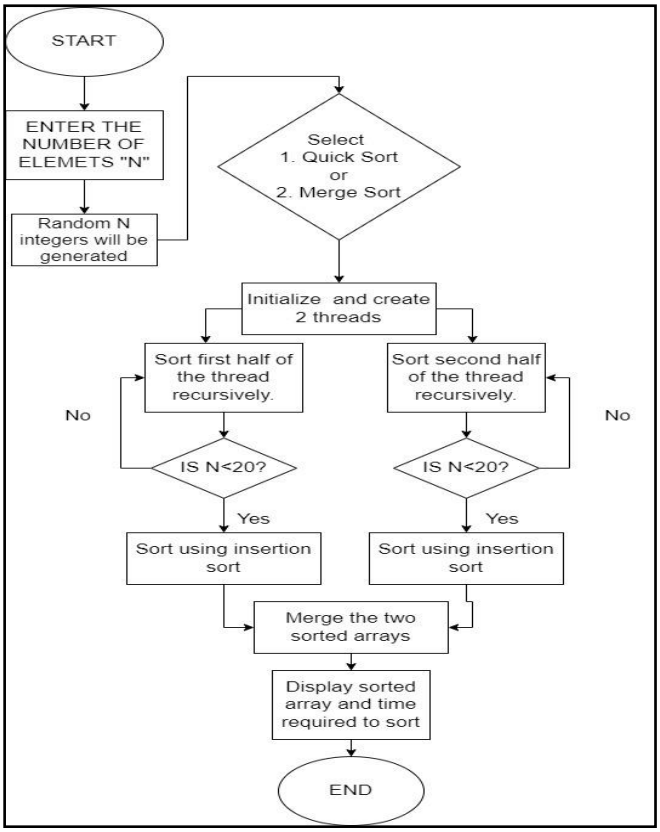

Figure 9

VI. IMPLEMENTATION

For the implementation of the above-mentioned software architecture we have defined a class which stores the length and pointer to the array. It also has member functions for merge sort and quick sort. Below is the pseudo code for the mentioned class.

```
class data_base:
        *list, length
        data_base        //constructor:
                list = malloc(length);
        merge_sort(data_base)
        quick_sort(data_base)
```

Below functionality is used to execute the above mentioned software architecture:

1. Prompt the user to enter the number of elements and get the length of elements to be used.
2. Declared an object of class *data_base* and used the constructor to allocate memory for the list of arrays.
3. Used srand function to seed the rand function.
4. Filled the array list with random integers using rand() API.
5. Prompt the user to enter the sorting option between quick sort and merge sort.
6. Before starting the sorting algorithm, initialized an auto variable with the present clock time.
7. Call the merge sort or quick sort member function as asked by the user

Quick sort

a. Declare 3 pthread_t identifiers and attributes respectively.
b. Initialize 3 attributes.
c. Create 2 threads (for sorting) and wait for their completion using pthread_join.
d. The function pointer attached will identify the thread using a static variable.
e. If the thread is 1 then sort the list from 0^{th} index to middle point index.
f. If the thread is 2 then sort the list from middle point + 1 index to the last element of the array.
g. Each of these implementations are the same as will be mentioned further.
h. For the quick sort algorithm, we need a pivot element and sort the left half of the pivot and right half of the pivot recursively.
i. From the reduced array, we are choosing the right most element of the array as the pivot element.
j. Once the pivot is chosen, our algorithm will place the pivot element at optimal position.
k. Similarly left half and right half of the pivot element will be sorted.
l. If any of the reduced array's size in the recursive functions falls below 20 (configurable), we directly used insertion sort algorithm.
m. Once the array is sorted, we exit the pthread
n. Create a 3^{rd} thread and use it to merge the two arrays in sorted manner.
o. For this we just compare each element of two arrays and whichever is smaller, place it into the resultant array
p. Once merged, exit the thread, returned the array and noted the time again at that moment.
q. Finally, we displayed the output array and the execution time,

Merge Sort

a. Declare 3 pthread_t identifiers and attributes respectively.
b. Initialize 3 attributes.
c. Create 2 threads (for sorting) and wait for their completion using pthread_join.
d. The function pointer attached will identify the thread using a static variable.
e. If the thread is 1 then sort the list from 0^{th} index to middle point index.
f. If the thread is 2 then sort the list from middle point + 1 index to the last element of the array.
g. Each of these implementations are the same as will be mentioned further.
h. For merge sort, we just keep on dividing the array into two halves.
i. Once we get the reduced array less then 20, then we used insertion sort.
j. The sorted array is then merged recursively.
k. Once the array is sorted, we exit the pthread
l. Create a 3rd thread and use it to merge the two arrays in sorted manner.
m. For this we just compare each element of two arrays and whichever is smaller, place it into the resultant array
n. Once merged, exit the thread, returned the array and noted the time again at that moment.
o. Finally, we displayed the output array and the execution time,

VII. Tests and Results

We used *"CMock"* [6] to test the code coverage and proper execution and flow of our program. Each line of code we design is testable, and the most efficient way to test it is through unit-tests. A unit test is a method of testing a unit, which is the smallest amount of code in a system that can be logically separated. That is a function, a subroutine, a method, or a property in most programming languages. CMock is a framework for generating mocks based on a header API. All you have to do to use CMock is add a mock header file to the test suite file.
For C functions, CMock generates mocks and stubs. It's useful for interaction-based unit testing, which involves determining how one module interacts with another. CMock assists you by constructing false versions of all the "other" modules, rather than attempting to compile them all together. You may then test your module's functionality using those false versions! CMock generates C source modules according to the interfaces defined in your C header files using Ruby scripts. It looks for function declarations in your header files. A unit-test framework has two fundamental concepts to grasp. The first is the unit-test framework itself, which only allows you to make assertions and write tests in a format that the framework can comprehend. Some frameworks allow you to "register" for tests, after which they will run all of the tests that have been registered. Other frameworks may employ self-registering macros or scripts to register and run your test methods at compile time. Unity allows you to execute tests in an organized manner.

Results:

A. Unit testing

Below are test case defined for merge sort

```
test_merge_sort_null_array(){
    TEST_ASSERT_EQUAL(0, merge_sort(NULL, 1, 5));
}

test_merge_sort_one_element(){
    int a[1]={1};
    TEST_ASSERT_EQUAL(1, merge_sort(a, 1, 1));
}

test_merge_sort_already_sorted(){
    int a[5]={1,2,3,4,5};
    TEST_ASSERT_EQUAL_INT_ARRAY(a, merge_sort(a, 1, 5), 5);
}

test_merge_sort_all_elements_equal(){
    int a[5]={1,1,1,1,1};
    TEST_ASSERT_EQUAL_INT_ARRAY(a, merge_sort(a, 1, 5), 5);
}

test_merge_sort_containing_duplicates(){
    int a[6]={10,3,22,10,33};
    int a1[6]={3,10,10,22,33};
    TEST_ASSERT_EQUAL_INT_ARRAY(a1, merge_sort(a, 1, 5), 5);
}
```

Figure 10

Below is the result of the above-mentioned unit test cases:

```
141   TEST_ASSERT_EQUAL_INT_ARRAY(a, merge_sort(a, 1, 5), 5);
142   }
143
144   test_merge_sort_all_elements_equal() {
145     int a[5] = {1, 1, 1, 1, 1};
146     TEST_ASSERT_EQUAL_INT_ARRAY(a, merge_sort(a, 1, 5), 5);
147   }
148
149   test_merge_sort_containing_duplicates() {
150     int a[6] = {10, 3, 22, 10, 33};
151     int a1[6] = {3, 10, 10, 22, 33};
152     TEST_ASSERT_EQUAL_INT_ARRAY(a1, merge_sort(a, 1, 5), 5);
153   }
154

PROBLEMS 2   OUTPUT   TERMINAL   GITLENS   COMMENTS   DEBUG CONSOLE

--- Executing [test_sorting.exe] ---

0 Tests 0 Failures 0 Ignored
OK

--- SUCCESS - Completed [test_sorting.exe] with error code [0] ---

--- Executing [test_line_buffer.exe] ---
```

Figure 11

B. Integrated Testing

Below are the test cases defined and ran for this project.

a. Number of elements used 40

b. Get random integers → PASS

c. Get prompt from user as quick sort / merge sort → PASS

d. Initialize and create two threads → PASS

e. Two threads should run concurrently → PASS (Over lapping prints of thread)

f. Exit the thread and merge the sorted arrays in sorted manner and print the results → PASS

d. Display the executed time → PASS

Below is the screenshot of the successful sorting:

1. Quick Sort

```
Enter the number of elements
40
Below are the random numbers generated
157     49      350     334     106     129     38      360     366     201     255     134     297     346     25
391     140     47      303     25      382     180     219     219     79      123     301     101     27      221
343     262     257     350     13      94      308     276     395     269
Press 1 for QuickSort
Press 2 for MergeSort
1
Quick Sort ==================>
Thread 1: 0-Thread 2: 21-39
20
13      25      25      27      38      47      49      79      94      101     106     123     129     134     140
157     180     201     219     219     221     255     257     262     269     276     297     301     303     308
334     343     346     350     350     360     366     382     391     395
Time taken 17502 microseconds

Process returned 0 (0x0)   execution time : 2.895 s
Press any key to continue.
```

Figure 12

2. Merge Sort

```
Enter the number of elements
40
Below are the random numbers generated
2       290     283     298     206     61      74      277     318     54      54      267     395     176     4
365     371     34      0       376     316     138     350     251     89      161     371     140     106     257
244     119     265     100     319     264     34      31      393     300
Press 1 for QuickSort
Press 2 for MergeSort
2
Merge Sort ==================>
Thread 1: 0-Thread 2: 21-39
20
0       2       4       31      34      34      54      54      61      74      89      100     106     119     138
140     161     176     206     244     251     257     264     265     267     277     283     290     298     300
316     318     319     350     365     371     371     376     393     395
Time taken 32137 microseconds

Process returned 0 (0x0)   execution time : 3.761 s
Press any key to continue.
```

Figure 13

CONCLUSIONS

After running multiple test cases and analyzing the results we have concluded that

- Multithreading allowed the execution of multiple parts of a program at the same time
- Multithreading lead to maximum utilization of the CPU by multitasking.

ACKNOWLEDGMENT

We would like to express our gratitude and thanks to Prof. Jahan Ghofraniha for giving us the opportunity to work on this project and guiding us with the concepts of Operating System throughout the course of this semester.

REFERENCES

[1] Abraham Silberschatz, Greg Gagne, Peter B. Galvin - Operating System Concepts

[2] https://www.javatpoint.com/multithreading-models-in-operating-system

[3] https://www.interviewbit.com/tutorial/quicksort-algorithm

[4] https://www.geeksforgeeks.org/quick-sort

[5] https://www.c-programming-simple-steps.com

[6] http://www.throwtheswitch.org/cmock

Members And Contribution

1. VILAS DHURI: Insertion sort, Execution time calculation and data base class definitions. Unit and Integrating Testing. Report making
2. RISHABH GUPTA: pThread architecture implementation and pointer functions use cases, Quick Sort algorithm implementation. Report making
3. SAHARSH SHIVHARE: Merge Sort algorithm implementation. Merging two sorted arrays in sorted fashion. Designing presentation. Report making

Appendix

Please find the source code and other project related docs below:
Github Link: https://github.com/grishabh895/Multithreaded-Sorting-Program
Source Code:

```
#include <iostream>
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <chrono>
using namespace std::chrono;
using namespace std;
void *runner_merge_sort(void *param); /*
threads call this function */

class data_base {
public:
    int *list;
    int length;
    int thread_number;
    data_base(int l) {
        length = l;
        list =
(int*)malloc(length*sizeof(int));
    }
    void set_list(int num, int index);
    int get_list(int index);
    void
merge_sort_multithreading(data_base data);
    void
quick_sort_multithreading(data_base data);
};
```

```
void data_base::set_list(int num, int
index) {
    list[index] = num;
}
int data_base::get_list(int index) {
   return list[index];
}

void merge(int arr[], int left, int
middle, int right) {

    int i, j, k;
    int half1 = middle - left + 1;
    int half2 = right - middle;

    int first[half1], second[half2]; //
temp arrays

    for (i = 0; i < half1; i++) {
        first[i] = arr[left + i];
    }

    for (j = 0; j < half2; j++) {
        second[j] = arr[middle + 1 + j];
    }

    i = 0;
    j = 0;
    k = left;

    while (i < half1 && j < half2) {

        if (first[i] <= second[j]) {
            arr[k] = first[i];
            ++i;
        } else {
            arr[k] = second[j];
            j++;
        }

        k++;
    }

    while (i < half1) {
        arr[k] = first[i];
        i++;
        k++;
    }

    while (j < half2) {
```

```
        arr[k] = second[j];
        j++;
        k++;
    }
}

// A utility function to swap two elements
void swap(int* a, int* b)
{
    int t = *a;
    *a = *b;
    *b = t;
}

/* This function takes last element as
pivot, places
the pivot element at its correct position
in sorted
array, and places all smaller (smaller
than pivot)
to left of pivot and all greater elements
to right
of pivot */
int partition (int arr[], int low, int
high)
{
    int pivot = arr[high]; // pivot
    int i = (low - 1);

    for (int j = low; j <= high - 1; j++)
    {
        // If current element is smaller
than the pivot
        if (arr[j] < pivot)
        {
            i++; // increment index of
smaller element
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

/* The main function that implements
QuickSort
arr[] --> Array to be sorted,
low --> Starting index,
high --> Ending index */
```

```
void quickSort(int arr[], int low, int
high)
{
    int arrsize = high - low + 1;
    if ((low < high) && arrsize <= 20)
    {
        // insertion sort algorithm
        for (unsigned int i = low; i <=
high; i++) {
            int temp = arr[i];
            unsigned int j = i;
            while (j > low && temp < arr[j
- 1]) {
                arr[j] = arr[j - 1];
                j--;
            }
            arr[j] = temp;
        }
    } else {
        if (low < high)
        {
            int pi = partition(arr, low,
high);
            quickSort(arr, low, pi - 1);
            quickSort(arr, pi + 1, high);
        }
    }
}

void *runner_quick_sort(void *param)
{
    data_base *data = (data_base*)param;
    int midpoint = data->length/2;
    static int thread = 0;
    thread++;
    if (thread == 1) {
        cout << "Thread 1: 0-" << midpoint
<<endl;
        quickSort(data->list, 0,
midpoint);
        pthread_exit(0);
    }

   if (thread == 2) {
        cout << "Thread 2: "<< midpoint +
1 << "-" << data->length - 1 <<endl;
        quickSort(data->list, midpoint +
1, data->length - 1);
        pthread_exit(0);
    }
```

```
    if (thread == 3) {
        merge(data->list, 0, (data-
>length/2), data->length-1);
        pthread_exit(0);
    }
}

void
data_base::quick_sort_multithreading(data_
base data) {
    pthread_t tid; /* the thread
identifier */
    pthread_attr_t attr; /* set of thread
attributes */

    pthread_t tid2; // second thread id
    pthread_attr_t attr2; // second thread
attributes

    pthread_t tid3; // third thread id
    pthread_attr_t attr3;

    pthread_attr_init(&attr); /* get the
default attributes */
    pthread_create(&tid,&attr,runner_quick
_sort, &data); /* create the thread */

    pthread_attr_init(&attr2);
    pthread_create(&tid2,&attr2,
runner_quick_sort, &data);
    pthread_join(tid, NULL);
    pthread_join(tid2, NULL);

    pthread_attr_init(&attr3);
    pthread_create(&tid3, &attr2,
runner_quick_sort, &data);
    pthread_join(tid3, NULL);

    int j;
    for (j = 0; j < data.length; j++) {
        if (j == data.length-1) {
            printf("%d\n", data.list[j]);
        } else {
            printf("%d\t", data.list[j]);
        }
    }

}
```

```
void merge_sort(int arr[], int l, int r) {
        int arrsize = r - l + 1;
    if ((l < r) && arrsize <= 20)
    {
        // insertion sort algorithm
        for (unsigned int i = l; i <= r;
i++) {
            int temp = arr[i];
            unsigned int j = i;
            while (j > l && temp < arr[j -
1]) {
                arr[j] = arr[j - 1];
                j--;
            }
            arr[j] = temp;
        }
    } else {
        if (l < r) {
            int middle = (l + (r-1))/ 2;
            merge_sort(arr, l, middle);
            merge_sort(arr, middle+1, r);
            merge(arr, l, middle, r);
        }
    }
}

void *runner_merge_sort(void *param)
{
    data_base *data = (data_base*)param;
    int midpoint = data->length/2;
    static int thread = 0;
    thread++;
    if (thread == 1) {
        cout << "Thread 1: 0-" << midpoint
<<endl;
        merge_sort(data->list, 0,
midpoint);
        pthread_exit(0);
    }

   if (thread == 2) {
        cout << "Thread 2: "<< midpoint +
1 << "-" << data->length - 1 <<endl;
        merge_sort(data->list, midpoint +
1, data->length - 1);
        pthread_exit(0);
    }

    if (thread == 3) {
```

```
        merge(data->list, 0, (data->length/2), data->length-1);
        pthread_exit(0);
    }


}

void data_base::merge_sort_multithreading(data_base data) {
    pthread_t tid; /* the thread identifier */
    pthread_attr_t attr; /* set of thread attributes */

    pthread_t tid2; // second thread id
    pthread_attr_t attr2; // second thread attributes

    pthread_t tid3; // third thread id
    pthread_attr_t attr3;

    clock_t t1, t2;
    t1 = clock();
    pthread_attr_init(&attr); /* get the default attributes */
    pthread_create(&tid,&attr,runner_merge_sort, &data); /* create the thread */

    pthread_attr_init(&attr2);
    pthread_create(&tid2,&attr2, runner_merge_sort, &data);
    pthread_join(tid, NULL);
    pthread_join(tid2, NULL);

    pthread_attr_init(&attr3);
    pthread_create(&tid3, &attr2, runner_merge_sort, &data);
    pthread_join(tid3, NULL);
    t2 = clock();

    int j;
    for (j = 0; j < data.length; j++) {
        if (j == data.length-1) {
            printf("%d\n", data.list[j]);
        } else {
            printf("%d\t", data.list[j]);
        }
    }

}
int main()
{
    int length, sorting_option;
    printf("Enter the number of elements \n");
    scanf("%d", &length);
    data_base data1(length);
    printf("Below are the random numbers generated \n");
    srand (time(NULL));
    for (int i =0; i<length ; i++) {
        data1.set_list(rand()%(10*length), i);
        printf("%d\t", data1.get_list(i));
    }
    printf("\nPress 1 for QuickSort\nPress 2 for MergeSort\n");

    scanf("%d", &sorting_option);
    auto start = high_resolution_clock::now();
    switch(sorting_option) {
    case 1:
        cout << "Quick Sort ====================>" <<endl;
        data1.quick_sort_multithreading(data1);
        break;
    case 2:
        cout << "Merge Sort ====================>" <<endl;
        data1.merge_sort_multithreading(data1);
        break;
    default:
        break;
    }
    auto stop = high_resolution_clock::now();
    auto duration = duration_cast<microseconds>(stop - start);
    cout << "Time taken "<< duration.count() << " microseconds" <<endl;
    return
```