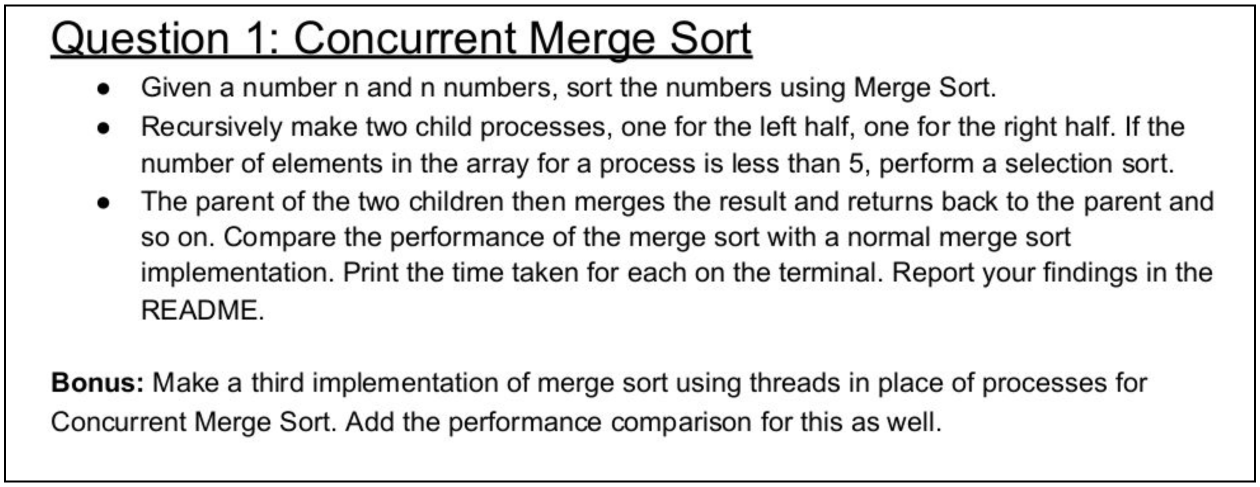
Report for Q1

Operating Systems

Kushagra Agarwal
2018113012



Question 1: Concurrent Merge Sort

- Given a number n and n numbers, sort the numbers using Merge Sort.
- Recursively make two child processes, one for the left half, one for the right half. If the number of elements in the array for a process is less than 5, perform a selection sort.
- The parent of the two children then merges the result and returns back to the parent and so on. Compare the performance of the merge sort with a normal merge sort implementation. Print the time taken for each on the terminal. Report your findings in the README.

Bonus: Make a third implementation of merge sort using threads in place of processes for Concurrent Merge Sort. Add the performance comparison for this as well.

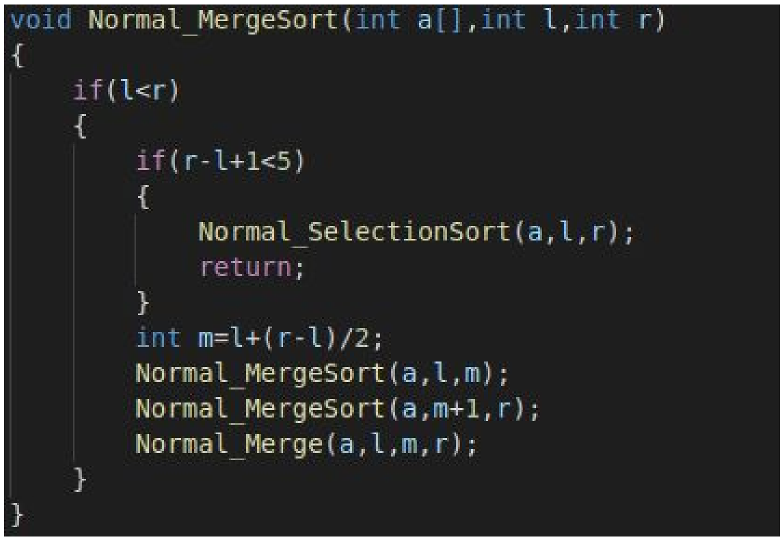
I implemented all three sorting algorithms

- Concurrent (Processes)
- Threaded (Threads)
- Normal

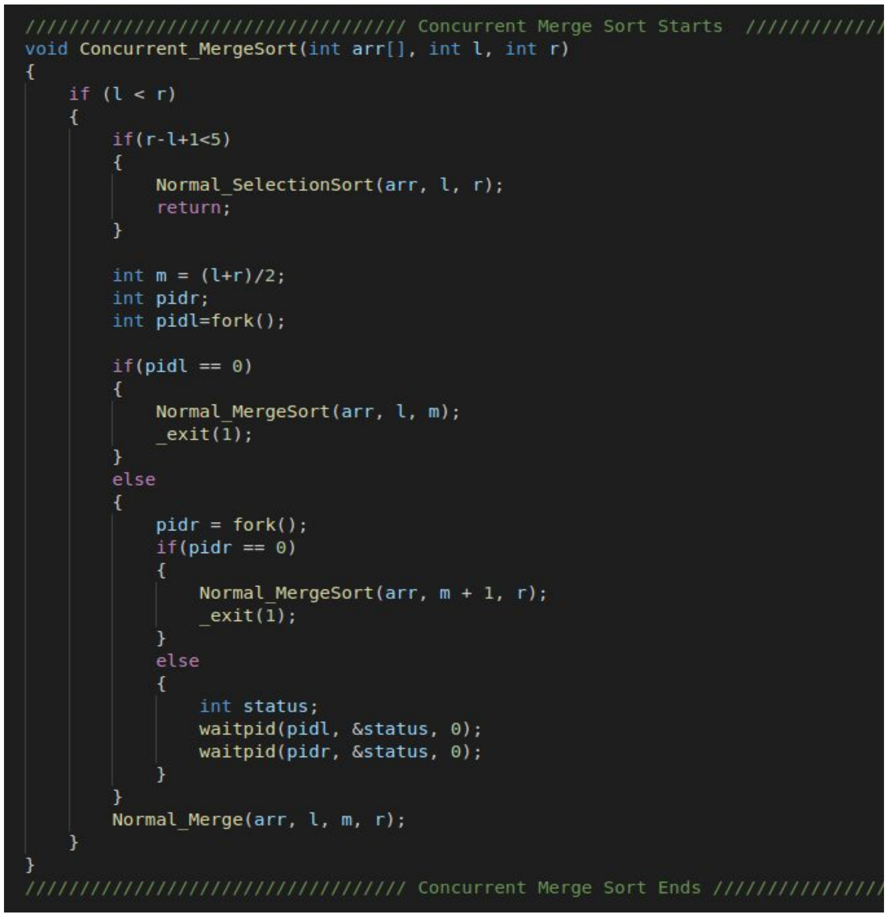
The code takes as input a number (defining the length of the array to be sorted). In the next line, it then takes the elements of the array.

Shared memory was created for the Processes implementation and Struct was created to be sent as a parameter input for the Threads implementation.

Merge Sort (Normal) was implemented using the Normal_MergeSort recursive function which called Normal_Merge to merge the sorted child arrays and Normal_SelectionSort to sort arrays of lengths less than 5. The code for Normal_MergeSort is given below:


```c
void Normal_MergeSort(int a[],int l,int r)
{
    if(l<r)
    {
        if(r-l+1<5)
        {
            Normal_SelectionSort(a,l,r);
            return;
        }
        int m=l+(r-l)/2;
        Normal_MergeSort(a,l,m);
        Normal_MergeSort(a,m+1,r);
        Normal_Merge(a,l,m,r);
    }
}
```

For Concurrent Processes, I used the idea of forking to create child processes to implement the recursion. I first forked and implemented left array sorting using the child process. In the parent process, I forked again and implemented the sorting of the right array in the child of this parent. The parent in the parent waits for the 2 processes to complete after which it merges the 2 sorted half-arrays. The code for the Concurrent_MergeSort driver function can be seen below. The same functions(Normal_Merge) and (Normal_SelectionSort) were used for logic implementation.

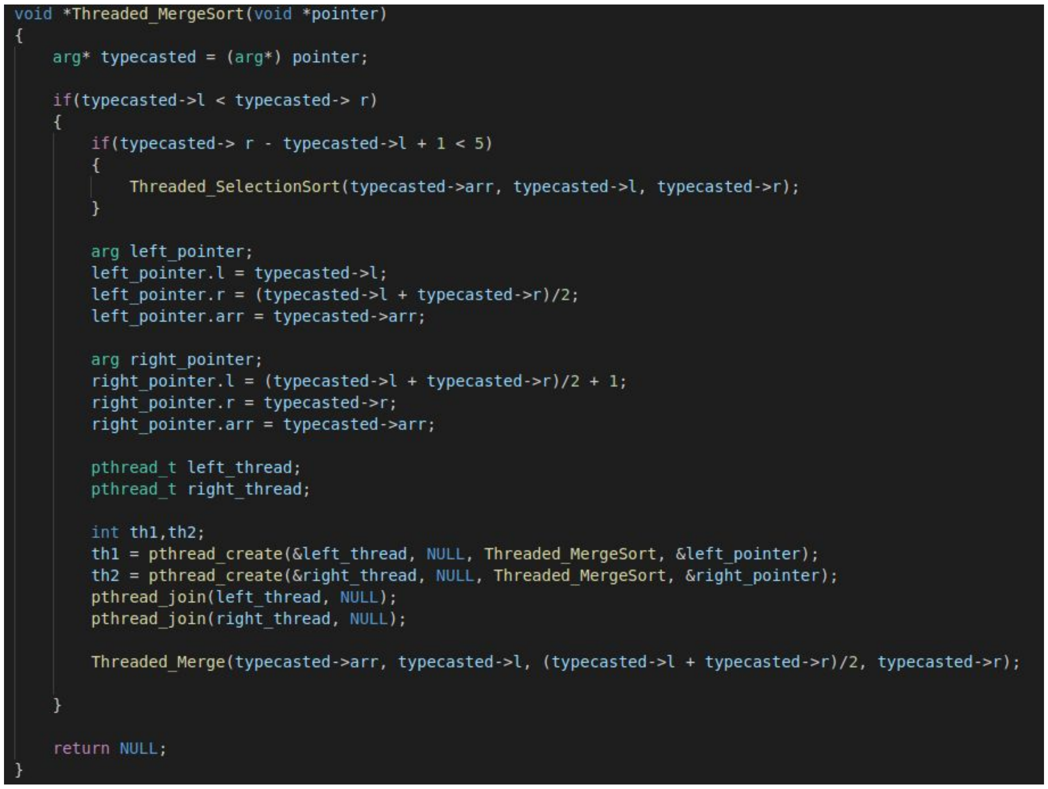


```
////////////////////////////////// Concurrent Merge Sort Starts  ////////////
void Concurrent_MergeSort(int arr[], int l, int r)
{
    if (l < r)
    {
        if(r-l+1<5)
        {
            Normal_SelectionSort(arr, l, r);
            return;
        }

        int m = (l+r)/2;
        int pidr;
        int pidl=fork();

        if(pidl == 0)
        {
            Normal_MergeSort(arr, l, m);
            _exit(1);
        }
        else
        {
            pidr = fork();
            if(pidr == 0)
            {
                Normal_MergeSort(arr, m + 1, r);
                _exit(1);
            }
            else
            {
                int status;
                waitpid(pidl, &status, 0);
                waitpid(pidr, &status, 0);
            }
        }
        Normal_Merge(arr, l, m, r);
    }
}
////////////////////////////////// Concurrent Merge Sort Ends ///////////////
```

For Threaded Merge Sort implementation the driver function Threaded_MergeSort was used. It basically created 2 threads and sent leftmost_index and rightmost_index to the half arrays along with the half arrays itself and called Threaded_Merge and Threaded_Selection on it. Threads take parameters as structs hence 2 structs were created to be sent as parameterized inputs to the threads. The code for Threaded_MergeSort can be found below.



```c
void *Threaded_MergeSort(void *pointer)
{
    arg* typecasted = (arg*) pointer;

    if(typecasted->l < typecasted-> r)
    {
        if(typecasted-> r - typecasted->l + 1 < 5)
        {
            Threaded_SelectionSort(typecasted->arr, typecasted->l, typecasted->r);
        }

        arg left_pointer;
        left_pointer.l = typecasted->l;
        left_pointer.r = (typecasted->l + typecasted->r)/2;
        left_pointer.arr = typecasted->arr;

        arg right_pointer;
        right_pointer.l = (typecasted->l + typecasted->r)/2 + 1;
        right_pointer.r = typecasted->r;
        right_pointer.arr = typecasted->arr;

        pthread_t left_thread;
        pthread_t right_thread;

        int th1,th2;
        th1 = pthread_create(&left_thread, NULL, Threaded_MergeSort, &left_pointer);
        th2 = pthread_create(&right_thread, NULL, Threaded_MergeSort, &right_pointer);
        pthread_join(left_thread, NULL);
        pthread_join(right_thread, NULL);

        Threaded_Merge(typecasted->arr, typecasted->l, (typecasted->l + typecasted->r)/2, typecasted->r);

    }

    return NULL;
}
```

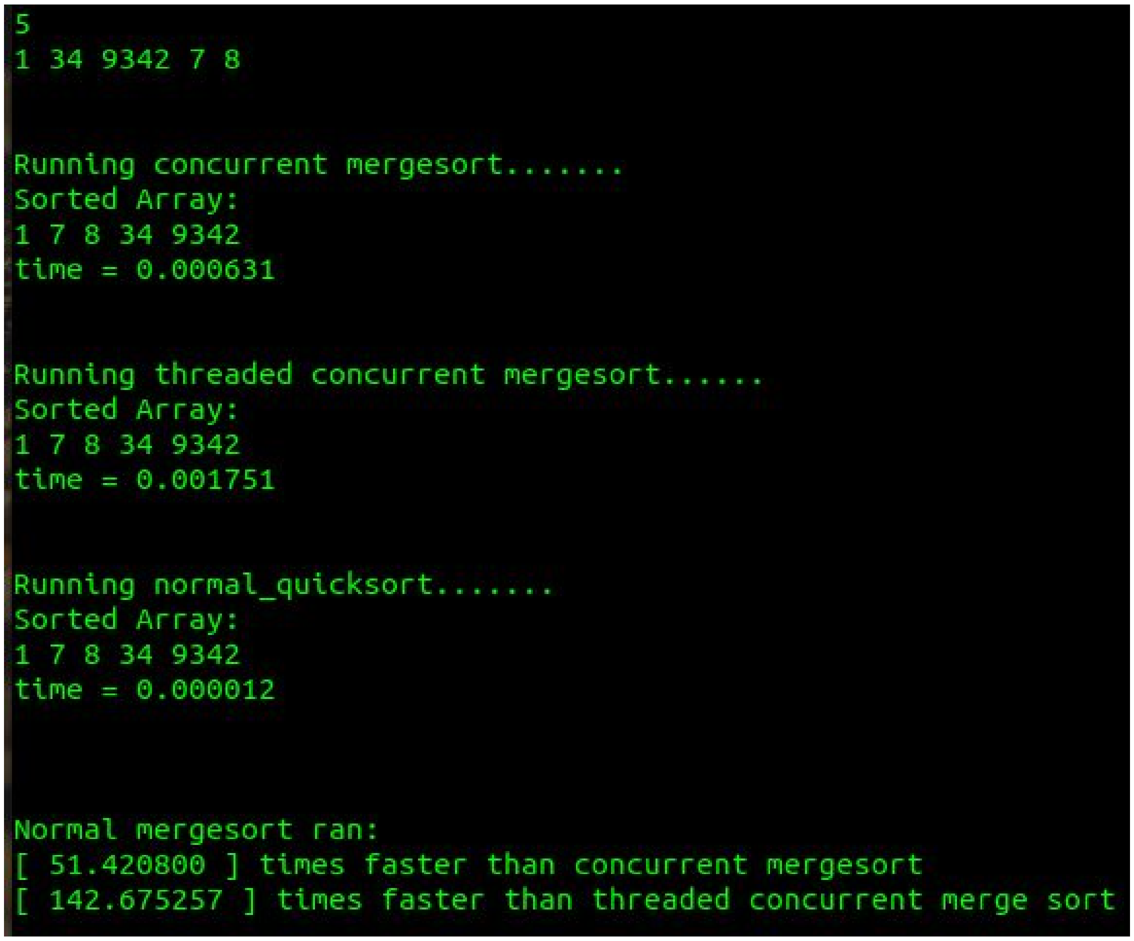
Time Comparison

Time taken by each of the three implementations were compared and the following output was generated by the code.

It can be clearly seen that the Normal merge sort is the fastest and the time taken by

- Concurrent merge sort is ~51 times more
- Threaded merge sort is ~142 times more

as compared to Normal merge sort.



```
5
1 34 9342 7 8


Running concurrent mergesort.......
Sorted Array:
1 7 8 34 9342
time = 0.000631


Running threaded concurrent mergesort......
Sorted Array:
1 7 8 34 9342
time = 0.001751


Running normal_quicksort.......
Sorted Array:
1 7 8 34 9342
time = 0.000012



Normal mergesort ran:
[ 51.420800 ] times faster than concurrent mergesort
[ 142.675257 ] times faster than threaded concurrent merge sort
```

Concurrent merge sort takes more time than the normal implementation as it requires a lot of processes to be created and therefore a lot of overtime is wasted on context switches etc.

Threaded merge sort takes more time than the normal merge sort as it requires the creation of threads which are then executing concurrently. This overtime can be avoided in the normal scenario.