Operating Systems

Homework Report

Muhammad Youshay (07103)

Assignment 4: Multi-threading

December 3, 2023

Table of Contents

1 Code Snippets –

1.0.1 Single Thread

Structures Used and Global Variables

```
// Structure to hold the dynamic array information which I have created
typedef struct {
    long long int *array_elements;
    size_t is_used;
    size_t size;
} myArray;
//Global variables and initializations
//LLONG_MAX and LLONG_MIN are macros that are defined in the limits.h
long long int sum = 0;
long long int min = LLONG_MAX;
long long int max = LLONG_MIN;
```

Function to insert and initialize the data that has been read from the input file

```
void initializeAndInsert(myArray *arr, FILE *file) {
    const size_t startCap = 1;
    //Dynamic memory allocation
    arr->array_elements = (long long int *)malloc(startCap * sizeof(long long
        int));
    //Checking if the memory is allocated or not
    if (arr->array_elements == NULL) {
        fprintf(stderr, "Cannot Allocate Memory\n");
        exit(EXIT_FAILURE);
    }
    arr->is_used = 0;
    arr->size = startCap;
    long long int currentElement;
    //Reading the file and inserting the elements into the array as long long
        int
    while (fscanf(file, "%lld", &currentElement) == 1) {
        if (arr->is_used == arr->size) {
            //Doubling the size of the array if the array is full. Following
                linked list concept
            size_t newSize = arr->size * 2;
            //Reallocating the memory
            long long int *newArray = (long long int
                *)realloc(arr->array_elements, newSize * sizeof(long long int));
            if (newArray == NULL) {
                fprintf(stderr, "Cannot Allocate Memory\n");
                exit(EXIT_FAILURE);
            }
```

```
            arr->array_elements = newArray;
            arr->size = newSize;
        }
        //Inserting the elements into the array
        arr->array_elements[arr->is_used] = currentElement;
        arr->is_used++;
    }
}
```

Function to free the dynamic memory when not in use

```
void freeMyArray(myArray *arr) {
    free(arr->array_elements);
    arr->array_elements = NULL;
    arr->is_used = arr->size = 0;
}
```

Main function where the input file is read. The sum, minimum and maximum is calculated. And the code is timed for comparison with multi thread program.

```
//Main function
int main(int argc, char *argv[]) {
    //Checking if the correct number of command line arguments are provided
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <program_name> <filename>\n", argv[0]);
        return EXIT_FAILURE;
    }
    const char *filename = argv[1];
    //Opening the file and reading
    FILE *file = fopen(filename, "r");
    if (file == NULL) {
        perror("Error opening file");
        return EXIT_FAILURE;
    }
    //Creating and populating the dynamic array from the file
    myArray arrayData;
    //Calling the function I created above
    initializeAndInsert(&arrayData, file);
    fclose(file);

    //Timing the processing time as per the code given in the assignment pdf
    clock_t start_time = clock();
    for (size_t i = 0; i < arrayData.is_used; i++) {
        sum += arrayData.array_elements[i];
        if (arrayData.array_elements[i] < min) {
            min = arrayData.array_elements[i];
        }
        if (arrayData.array_elements[i] > max) {
            max = arrayData.array_elements[i];
        }
```

```
    }
    clock_t end_time = clock();
    double elapsed_time = ((double)(end_time - start_time)) / CLOCKS_PER_SEC;

    // Printing the results and processing time
    printf("Sum: %lld\n", sum);
    printf("Minimum: %lld\n", min);
    printf("Maximum: %lld\n", max);
    printf("Time taken - Main - : %f seconds\n", elapsed_time);

    // Freeing the memory occupied by the dynamic array by calling the
       function we created above
    freeMyArray(&arrayData);
    return EXIT_SUCCESS;
}
```

1.0.2 Multi Thread

The myArray structure is the same as the single threaded program. However, an additional structure has been created for each thread and some additional global variables have been added.

```
//Structure to hold information for each thread
typedef struct {
    myArray *dataArray;
    size_t start;
    size_t end;
    double thread_time;
} ThreadData;

//Global variables and initializations
//LLONG_MAX and LLONG_MIN are macros that are defined in the limits.h
long long int sum = 0;
long long int min = LLONG_MAX;
long long int max = LLONG_MIN;
pthread_mutex_t sum_lock;
pthread_mutex_t min_max_lock;
```

Initialize and Insert, and the freeMyArray functions remain the same. However, an additional function has been added to process each thread and calculate the sum, min and max for each thread and then use locks to update the global sum, min and max to ensure synchronization. We are also timing each thread in this function according to the code that was provided to us in the pdf.

```
void* processArrayChunk(void *arg) {
    ThreadData *dataThread = (ThreadData*)arg;
    struct timespec t_start, t_end;
    // Recording the current time at the start of the thread's execution. This
```

```
        is the same code given in the assignment pdf
    clock_gettime(CLOCK_MONOTONIC, &t_start);
    // Initializing the local variables to store the sum, min, and max of each
        thread
    long long int local_sum = 0, local_min = LLONG_MAX, local_max = LLONG_MIN;
    for (size_t i = dataThread->start; i < dataThread->end; i++) {
        local_sum += dataThread->dataArray->array_elements[i];
        if (dataThread->dataArray->array_elements[i] < local_min) {
            local_min = dataThread->dataArray->array_elements[i];
        }
        if (dataThread->dataArray->array_elements[i] > local_max) {
            local_max = dataThread->dataArray->array_elements[i];
        }
    }
    // Recording the current time at the end of the thread's execution
    clock_gettime(CLOCK_MONOTONIC, &t_end);
    // Combining the local sum, min, and max with the global sum, min, and max
    // We need to use mutex locks to ensure that the global variables are not
        being accessed by multiple threads at the same time
    pthread_mutex_lock(&sum_lock);
    sum += local_sum;
    pthread_mutex_unlock(&sum_lock);
    pthread_mutex_lock(&min_max_lock);
    if (local_min < min) min = local_min;
    if (local_max > max) max = local_max;
    pthread_mutex_unlock(&min_max_lock);
    // Calculating the thread's execution time and storing it in the
        ThreadData structure
    dataThread->thread_time = (t_end.tv_sec - t_start.tv_sec);
    dataThread->thread_time += (t_end.tv_nsec - t_start.tv_nsec) /
        1000000000.0;
    return NULL;
}
```

Main function

```
int main(int argc, char *argv[]) {
    myArray dataArray;
    struct timespec start, finish;
    double elapsed;
    // Checking if the correct number of command line arguments are provided
    if (argc < 2 || argc > 3) {
        fprintf(stderr, "Usage: %s <program_name> <filename> [numThreads]\n",
            argv[0]);
        return EXIT_FAILURE;
    }
    // Number of threads to use - if not provided default to 4
    size_t numThreads = (argc == 3) ? atoi(argv[2]) : 4;
    //Opening the file and reading
    const char *filename = argv[1];
    FILE *file = fopen(filename, "r");
```

```
if (file == NULL) {
    perror("Error opening file");
    return EXIT_FAILURE;
}
// Creating and populating the dynamic array from the file
initializeAndInsert(&dataArray, file);
fclose(file);
// Ensuring the provided number of threads is valid
if (numThreads <= 0 || numThreads > dataArray.is_used) {
    fprintf(stderr, "Invalid number of threads\n");
    return EXIT_FAILURE;
}
//Initializing the mutex locks
if (pthread_mutex_init(&sum_lock, NULL) != 0 ||
    pthread_mutex_init(&min_max_lock, NULL) != 0) {
    fprintf(stderr, "Mutex initialization failed\n");
    return EXIT_FAILURE;
}
// Dividing the array into equal-sized chunks for each thread
size_t chunkSize = dataArray.is_used / numThreads;
//Creating threads
pthread_t threads[numThreads];
ThreadData threadData[numThreads];
// Recording the current time at the start of the program's execution -
    mainr
clock_gettime(CLOCK_MONOTONIC, &start);
for (size_t i = 0; i < numThreads; i++) {
    threadData[i].dataArray = &dataArray;
    threadData[i].start = i * chunkSize;
    threadData[i].end = (i == numThreads - 1) ? dataArray.is_used : (i +
        1) * chunkSize;
    pthread_create(&threads[i], NULL, processArrayChunk, &threadData[i]);
}
// Waiting for all threads to finish and then joining them
for (size_t i = 0; i < numThreads; i++) {
    pthread_join(threads[i], NULL);
    // After joining each thread, print its execution time
    printf("Thread %zu time taken: %f seconds\n", i,
        threadData[i].thread_time);
}
clock_gettime(CLOCK_MONOTONIC, &finish);
elapsed = (finish.tv_sec - start.tv_sec);
elapsed += (finish.tv_nsec - start.tv_nsec) / 1000000000.0;
// Printing the results
printf("Sum: %lld\n", sum);
printf("Minimum: %lld\n", min);
printf("Maximum: %lld\n", max);
printf("Time taken for threads Execution - from Main Fuc - : %f
    seconds\n", elapsed);
// Freeing the memory occupied by the dynamic array
freeMyArray(&dataArray);
```

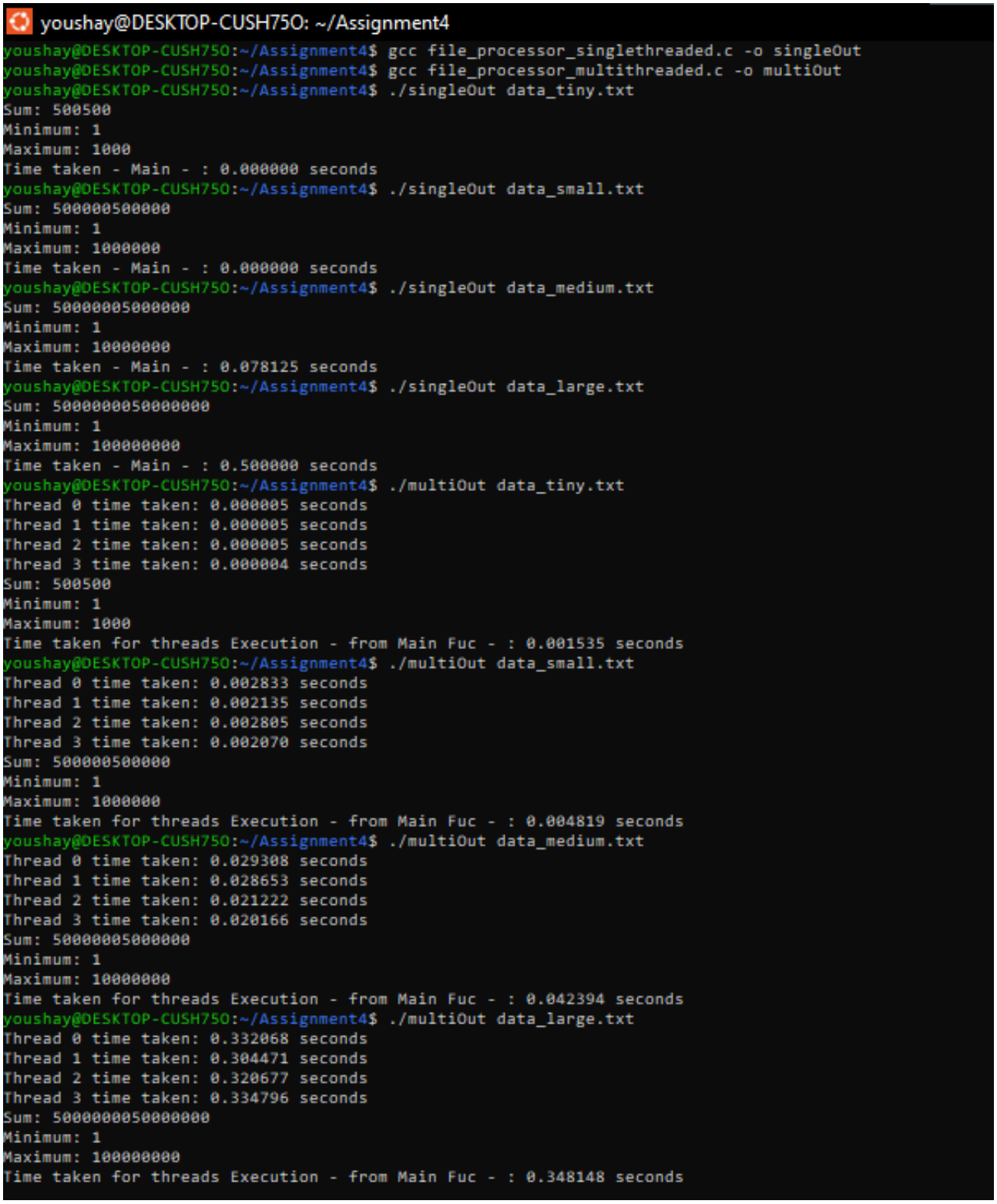
```
    //Destroying the mutex locks
    pthread_mutex_destroy(&sum_lock);
    pthread_mutex_destroy(&min_max_lock);
    return EXIT_SUCCESS;
}
```

2 MakeFile

Make file for running has been added to the zip file. Use the commands 'make build' and then 'make run' to run the generated output file. Once you are inside the .out file, you'll have to insert the input.

```
build:
	gcc file_processor_singlethreaded.c -o singleOut
	gcc file_processor_multithreaded.c -o multiOut
run:
	./singleOut
	./multiOut
clean:
	rm ./singleOut
	rm ./multiOut
rebuild: clean build
```

3 Outputs And Comparison



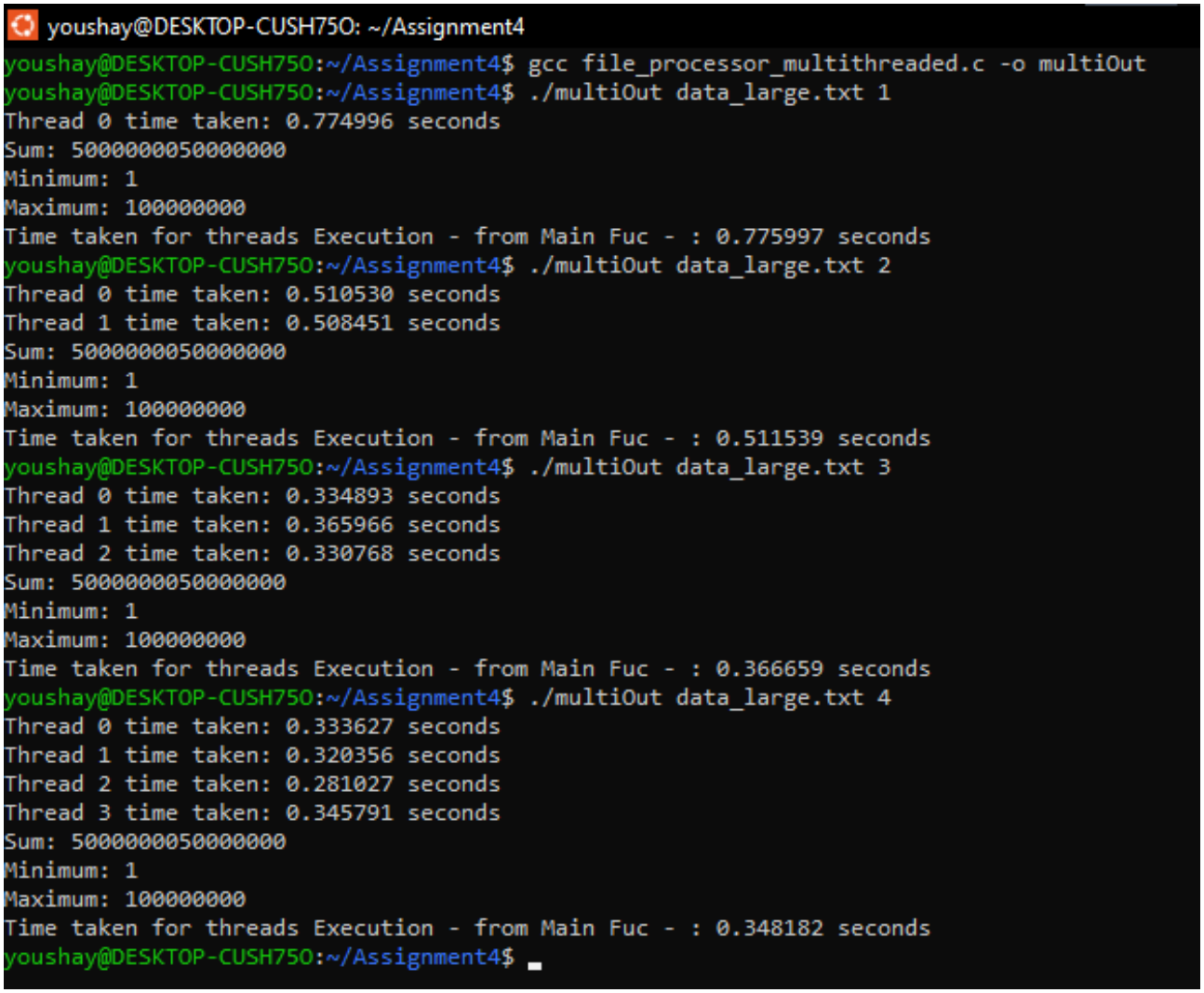
3.0.1 Average Timings Chart

SINGLE THREADED				
Run No	Tiny	Small	Medium	Large
1	0.0000	0.0000	0.078	0.5
2	0.0000	0.0000	0.076	0.508
3	0.0000	0.0000	0.075	0.52
4	0.0000	0.0000	0.074	0.517
AVERAGE	0.0000	0.0000	0.07575	0.51125

MULTI THREADED (4 Threads)				
Run No	Tiny	Small	Medium	Large
1	0.00153	0.0048	0.042	0.348
2	0.00152	0.0045	0.045	0.33
3	0.00161	0.0039	0.041	0.239
4	0.00143	0.0044	0.047	0.257
AVERAGE	0.0015225	0.0044	0.04375	0.2935

3.0.2 Analysis

To test our code, we first ran our single threaded program n all the three files and we say that the tiny-data file took the least time while the large-data file took the most time as expected. Then we ran our multi threaded program. We didn't give any number of threads so by default our program ran on 4 threads. For the tiny-data file the timings are almost similar as single threaded. For small-data, we have the same case as these timings are almost negligible. For data medium we see that our multi threaded program performs better than the single threaded program and same is the case with large-data where our multi-threaded program outperforms the single threaded program. These results are as per theory where a multi threaded program runs faster than the single threaded program. The sum, min and max values are coming out to be accurate for both.



To further test the multi threaded program I tested the code on different number of threads starting from one thread and going four threads. It can be clearly seen that as we increase the number of threads, the time reduces as it should.