Habib University
Operating Systems - CS232

Homework 04 - Report

Multi-threading

Instructor: Munzir Zafar

Sadiqah Mushtaq - 07152

Contents

1 Introduction **3**

2 Makefile **3**

3 Input **3**

4 Output **4**

5 Structures and Algorithms **4**

5.1 Single Threaded 4

5.2 Multi-threaded 7

6 Timing Comparison and Analysis **10**

6.1 Single-Threaded Program 11

6.1.1 Analysis 11

6.2 Multi-Threaded Program 11

6.2.1 Analysis 12

7 Overall Analysis: **13**

1 Introduction

This focuses on implementing a multi-threaded file processor using the POSIX Threads API in the C programming language. In this assignment, the goal is to efficiently process a large dataset concurrently, utilizing synchronization mechanisms to ensure thread safety and optimize resource utilization. The assignment challenges students to read a dataset from a file, perform various computations on it, and compare the performance of a single-threaded approach with a multi-threaded implementation.

2 Makefile

```
CC=gcc
CFLAGS=-Wall -Wextra -Wno-long-long -Wno-overflow

build_st:
  $(CC) $(CFLAGS) -o ST ST.c
build_mt:
  $(CC) $(CFLAGS) -o MT MT.c

run_st:
  ./ST Sadiqah data_tiny.txt
  ./ST Sadiqah data_small.txt
  ./ST Sadiqah data_medium.txt
  ./ST Sadiqah data_large.txt
run_mt:
  ./MT Sadiqah data_tiny.txt 4
  ./MT Sadiqah data_tiny.txt 10
  ./MT Sadiqah data_tiny.txt 25
  ./MT Sadiqah data_tiny.txt 50
  ./MT Sadiqah data_tiny.txt 100

clean:
  rm -f ST
  rm -f MT
```

The Makefile provides a set of targets for building, running, and cleaning up the project. The "build" target compiles the code and generates an executable named "Scheduler." The "run" target executes the compiled program, while the "clean" target removes the executable to maintain a clean project directory.

3 Input

The input for this assignment primarily consists of command-line arguments, enabling users to specify the program name, the data file to be processed, and, in the case of the multi-threaded program, the number of threads to launch. For the single-threaded program, the second command-line argument is the data file. The program performs error handling to ensure that the correct number of arguments is provided, guiding the user if there's any omission. The data files contain linear lists of integers ranging from 1 to 1,000,000 for "data_small.txt," 1 to 10,000,000 for "data_medium.txt," and 1 to 100,000,000 for "data_large.txt." The flexibility of specifying the data file allows for scalability and testing across different datasets. Sample Input:

```
./ST Sadiqah data_tiny.txt
./MT Sadiqah data_tiny.txt 12
```

4 Output

The output of the program includes the computed sum, minimum, and maximum values of the dataset. For the single-threaded program, the results are printed directly to the output console. In contrast, the multi-threaded program involves more intricate synchronization mechanisms to calculate these values across multiple threads. The final results are printed to the console, showcasing the effectiveness of the multi-threaded approach in concurrently processing the dataset. Additionally, the program prints the average elapsed time for both single-threaded and multi-threaded executions, providing insights into the performance gains achieved through parallelization. The output serves as a comprehensive evaluation of the program's efficiency in handling large datasets using multi-threading.

Sample Output for Single Threaded:

```
./ST Sadiqah data_tiny.txt
Program Name: Sadiqah
Minimum: 1
Maximum: 1000
Sum: 500500
Average Elapsed Time for Min/Max: 0.000008 seconds
Average Elapsed Time for Sum: 0.000008 seconds
```

Sample Output for Multi-threaded:

```
./MT Sadiqah data_tiny.txt 4
Average elapsed time for Sum in seconds is 0.000497
Average elapsed time for Min/Max in seconds is 0.000302
Program Name: Sadiqah
Total Sum: 500500
Minimum: 1
Maximum: 1000
```

In the single-threaded output, the program processes a dataset using a single thread, yielding calculated values and elapsed time.
The multi-threaded output, run on the same dataset with multiple threads, shows individual timings for each thread and provides aggregated results. The threaded execution demonstrates the concurrent processing of data, leading to potential performance improvements. The variability in thread timings is expected due to the parallel nature of execution.

5 Structures and Algorithms

5.1 Single Threaded

```c
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

#define MAX_FILENAME_LENGTH 256
#define LLONG_MAX -1000000000
#define LLONG_MIN 1000000000

int main(int argc, char *argv[]) {
    long long int final_sum;
```

```c
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <program_name> <filename>\n", argv[0]);
        return 1;
    }

    char *programName = argv[1];
    char *filename = argv[2];

    long long int minimum = LLONG_MIN;
    long long int maximum = LLONG_MAX;
    long long int sum = 0;

    // Define variables for timing
    clock_t start_time, end_time;
    double elapsed_time_minmax = 0.0;
    double elapsed_time_sum = 0.0;

    FILE *file = fopen(filename, "r");
    if (!file) {
        perror("Error opening file");
        return 1;
    }

    long long int value;

     // Count the number of long long integers in the file
    size_t count = 0;
    while (fscanf(file, "%lld", &value) == 1) {
        // printf("Value: %lld\n", value);
        count++;
    }

    // Allocate memory for the dynamic array
    long long int *dataArray = (long long int *)malloc(count * sizeof(long long int));
    if (!dataArray) {
        perror("Memory allocation error");
        fclose(file);
        return 1;
    }

    // Reset file pointer to the beginning of the file
    fseek(file, 0, SEEK_SET);

    // Read data from the file into the dynamic array
    for(size_t i = 0; i < count; i++) {
        fscanf(file, "%lld", &dataArray[i]);
    }


    for (int j = 0; j < 5; j++) {
        // Start the timer for min_max
        start_time = clock();
        // Read data from the file into the dynamic array for min_max
        for (size_t i = 0; i < count; i++) {
            if (dataArray[i] < minimum) {
                minimum = dataArray[i];
            }
            if (dataArray[i] > maximum) {
                maximum = dataArray[i];
            }
        }
```

```

        // Stop the timer for min_max
        end_time = clock();
        elapsed_time_minmax += (double)(end_time - start_time) / CLOCKS_PER_SEC;

        // Start the timer for sum
        start_time = clock();
        // Read data from the file into the dynamic array for sum
        for (size_t i = 0; i < count; i++) {
            sum += dataArray[i];
        }

        // Stop the timer for sum
        end_time = clock();
        elapsed_time_sum += (double)(end_time - start_time) / CLOCKS_PER_SEC;

        if(j == 0) final_sum = sum;

    }
    free(dataArray);
    fclose(file);


    // Calculate average times
    double avg_elapsed_time_minmax = elapsed_time_minmax / 5;
    double avg_elapsed_time_sum = elapsed_time_sum / 5;

    // Print the results
    printf("Program Name: %s\n", programName);
    printf("Minimum: %lld\n", minimum);
    printf("Maximum: %lld\n", maximum);
    printf("Sum: %lld\n", final_sum);
    printf("Average Elapsed Time for Min/Max: %f seconds\n",
    avg_elapsed_time_minmax);
    printf("Average Elapsed Time for Sum: %f seconds\n", avg_elapsed_time_sum);

    return 0;
}
```

In the context of the single-threaded file processor program, the primary data structure employed is a dynamically allocated array (dataArray) to store the dataset read from the input file. This array is allocated based on the count of long long integers present in the file, providing a memory-efficient solution that adapts to the size of the dataset. The program uses this array to efficiently process and analyze the dataset, computing the minimum, maximum, and sum values in a single pass. The use of dynamically allocated memory allows for flexibility in handling datasets of varying sizes, optimizing resource utilization.

Additionally, scalar variables such as minimum, maximum, and sum are utilized to keep track of the minimum, maximum, and sum values, respectively. These variables are updated during the traversal of the dataset array. The chosen data structures align with the program's objective of performing simple statistical computations on a dataset, emphasizing memory efficiency and ease of access. The file handling is facilitated by the standard FILE structure in C, specifically using the fopen and fclose functions for file operations.

This straightforward approach ensures that the program efficiently manages memory, processes the dataset, and computes the required statistical values within a single-threaded execution context. In the subsequent multi-threaded version, modifications and additional data structures will be introduced to enable parallel processing while maintaining thread safety.

5.2 Multi-threaded

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

#define MAX_FILENAME_LENGTH 256
#define NUM_THREADS_DEFAULT 4

long long int finalSum = 0, finalMin = LLONG_MAX, finalMax = LLONG_MIN;
pthread_mutex_t sumMutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t minMaxMutex = PTHREAD_MUTEX_INITIALIZER;

typedef struct ThreadData {
    int threadId;
    long long int* dataArray;
    size_t start;
    size_t end;
} ThreadData;

void* sumFunction(void* arg) {
    ThreadData* threadData = (ThreadData*)arg;
    long long int partialSum = 0;

    for (size_t i = threadData->start; i < threadData->end; i++) {
        partialSum += threadData->dataArray[i];
    }

    pthread_mutex_lock(&sumMutex);
    finalSum += partialSum;
    pthread_mutex_unlock(&sumMutex);

    // printf("Thread %d: partial sum = %lld\n", threadData->threadId, partialSum);

    pthread_exit(NULL);
}

void* minMaxFunction(void* arg) {
    ThreadData* threadData = (ThreadData*)arg;
    long long int partialMin = LLONG_MAX, partialMax = LLONG_MIN;

    for (size_t i = threadData->start; i < threadData->end; i++) {
        if (threadData->dataArray[i] < partialMin) {
            partialMin = threadData->dataArray[i];
        }

        if (threadData->dataArray[i] > partialMax) {
            partialMax = threadData->dataArray[i];
        }
    }

    pthread_mutex_lock(&minMaxMutex);

    if (partialMin < finalMin) {
        finalMin = partialMin;
    }

    if (partialMax > finalMax) {
```

```c
        finalMax = partialMax;
    }

    pthread_mutex_unlock(&minMaxMutex);

    // printf("Thread %d: partial min = %lld, partial max = %lld\n", threadData->threadId, partialMin, partialMax);

    pthread_exit(NULL);
}

int main(int argc, char* argv[]) {
    long long int finalised_sum;
    char programName[MAX_FILENAME_LENGTH];
    char filename[MAX_FILENAME_LENGTH];
    int numThreads = NUM_THREADS_DEFAULT;

    if (argc < 3) {
        fprintf(stderr, "Number of arguments for %s are less than 3\n", argv[0]);
        return 1;
    }

    snprintf(programName, sizeof(programName), "%s", argv[1]);
    snprintf(filename, sizeof(filename), "%s", argv[2]);

    if (argc >= 4) {
        numThreads = atoi(argv[3]);
    } else {
        numThreads = NUM_THREADS_DEFAULT;
    }

    FILE* file = fopen(filename, "r");
    if (!file) {
        perror("Error opening file");
        return 1;
    }

    long long int value;
    size_t dataSize = 0;
    while (fscanf(file, "%lld", &value) == 1) {
        dataSize++;
    }

    long long int* dataArray = (long long int*)malloc(dataSize * sizeof(long long int));
    if (!dataArray) {
        perror("Memory allocation error");
        fclose(file);
        return 1;
    }

    fseek(file, 0, SEEK_SET);

    for (size_t i = 0; i < dataSize; i++) {
        fscanf(file, "%lld", &dataArray[i]);
    }

    fclose(file);

    struct timespec start_time_sum, end_time_sum, start_time_minmax, end_time_minmax;

```

```c
    double elapsed_time_sum = 0.0;
    double elapsed_time_minmax = 0.0;

    for (int j = 0; j < 5; j++) {
        size_t chunkSize = dataSize / numThreads;
        size_t remainder = dataSize % numThreads;
        size_t startIndex = 0;

        pthread_t sumThreads[numThreads];
        ThreadData sumThreadData[numThreads];

        pthread_t minMaxThreads[numThreads];
        ThreadData minMaxThreadData[numThreads];

        clock_gettime(CLOCK_MONOTONIC, &start_time_sum);

        for (int i = 0; i < numThreads; i++) {
            sumThreadData[i].dataArray = dataArray;
            sumThreadData[i].start = startIndex;
            sumThreadData[i].end = startIndex + chunkSize;
            if ((size_t)i < remainder) {
                sumThreadData[i].end += 1;
            }
            sumThreadData[i].threadId = i;

            if (pthread_create(&sumThreads[i], NULL, sumFunction, (void*)&
    sumThreadData[i]) != 0) {
                perror("Error: Thread was not created properly");
                free(dataArray);
                exit(1);
            }

            startIndex += sumThreadData[i].end - sumThreadData[i].start;
        }

        for (int i = 0; i < numThreads; i++) {
            pthread_join(sumThreads[i], NULL);
        }

        clock_gettime(CLOCK_MONOTONIC, &end_time_sum);

        clock_gettime(CLOCK_MONOTONIC, &start_time_minmax);

        startIndex = 0;
        for (int i = 0; i < numThreads; i++) {
            minMaxThreadData[i].dataArray = dataArray;
            minMaxThreadData[i].start = startIndex;
            minMaxThreadData[i].end = startIndex + chunkSize;
            if ((size_t)i < remainder) {
                minMaxThreadData[i].end += 1;
            }
            minMaxThreadData[i].threadId = i;

            if (pthread_create(&minMaxThreads[i], NULL, minMaxFunction, (void*)&
    minMaxThreadData[i]) != 0) {
                perror("Error: Thread was not created properly");
                free(dataArray);
                exit(1);
            }

            startIndex += minMaxThreadData[i].end - minMaxThreadData[i].start;
        }
```

```c

        for (int i = 0; i < numThreads; i++) {
            pthread_join(minMaxThreads[i], NULL);
        }

        clock_gettime(CLOCK_MONOTONIC, &end_time_minmax);

        elapsed_time_sum += (end_time_sum.tv_sec - start_time_sum.tv_sec) +
                            (end_time_sum.tv_nsec - start_time_sum.tv_nsec) / 1e9;

        elapsed_time_minmax += (end_time_minmax.tv_sec - start_time_minmax.tv_sec) +
                               (end_time_minmax.tv_nsec - start_time_minmax.tv_nsec) / 1e9;
        if (j == 0)finalised_sum = finalSum;
    }

    printf("Average elapsed time for Sum in seconds is %f\n", elapsed_time_sum / 5);
    printf("Average elapsed time for Min/Max in seconds is %f\n", elapsed_time_minmax / 5);

    printf("Program Name: %s\n", programName);
    printf("Total Sum: %lld\n", finalised_sum);
    printf("Minimum: %lld\n", finalMin);
    printf("Maximum: %lld\n", finalMax);

    free(dataArray);

    return 0;
}
```

The multithreaded C program, designed for concurrent processing of a substantial dataset, employs the POSIX Threads API for efficient parallelization. The code features a ThreadData structure, encapsulating essential information for each thread, and a core thread_function responsible for processing assigned data chunks while measuring execution time. In the main function, command-line arguments are handled, the file is read, and memory is dynamically allocated for the dataset. Threads are initialized, each assigned a portion of the dataset, and concurrent execution is achieved. The main thread joins the thread, calculates total statistics, and measures the overall execution time. Noteworthy is the inclusion of thread safety measures, error handling for file operations and memory allocation, and precision in time measurement using clock_gettime. This comprehensive approach ensures efficient parallel processing, synchronization, and robust performance evaluation in the context of multithreading.

6 Timing Comparison and Analysis

The experiments were conducted on an 2-core, 4-thread Intel i7-5600HQ CPU running at 2.60GHz. It's important to note that system configurations may introduce variations in performance.

Four distinct datasets were utilized for the evaluations:

- `data_tiny` (1,000 integers)
- `data_small` (1,000,000 integers)
- `data_medium` (1,000,000 integers)
- `data_large` (100,000,000 integers)

For each dataset, both single-threaded and multi-threaded programs were executed. The multi-threaded program varied the number of threads through command line arguments. The resulting averages, computed from 5 runs for each program, offer insights into overall performance.

6.1 Single-Threaded Program

The table below illustrates the average elapsed time for each dataset under the single-threaded program.

Dataset	Avg time(SUM)	Avg time(MIN_MAX)
`data_tiny`	0.000004	0.000005
`data_small`	0.008448	0.006678
`data_medium`	0.098092	0.119004
`data_large`	0.081217	0.099780

Table 1: Timings for Multi-Threaded Program on various numbers of threads

6.1.1 Analysis

Execution Time:

- The execution times for the single-threaded program are generally quite low, ranging from microseconds to milliseconds.
- The execution time increases as the size of the dataset increases, which is expected.

Observations:

- The smallest dataset, `data_tiny`, has consistently low execution times.
- The larger datasets, `data_medium` and `data_large`, have slightly higher execution times, which is typical for larger datasets.

6.2 Multi-Threaded Program

The table below summarizes the average time taken by a thread and the average elapsed time of the multi-threaded program for various numbers of threads for each dataset.

Tiny (1K Integers)			Small (1M Integers)		
# Threads	Avg t(SUM)	Avg t(MIN_MAX)	# Threads	Avg t(SUM)	Avg t(MIN_MAX)
4	0.000497	0.000302	4	0.004497	0.004159
10	0.000952	0.001191	10	0.006209	0.006286
25	0.008531	0.002823	10	0.009271	0.006063
50	0.011213	0.012378	50	0.011819	0.007248
100	0.021223	0.021280	100	0.027033	0.018296
Medium (10M Integers)			Large (100M Integers)		
# Threads	Avg t(SUM)	Avg t(MIN_MAX)	#Threads	Time/Thread	Elapsed Time
4	0.028334	0.055789	4	0.07294	0.085563
10	0.030391	0.065313	10	0.054789	0.086321
25	0.033003	0.055393	10	0.061873	0.09923
50	0.043972	0.055619	50	0.061273	0.083742
100	0.039843	0.064424	100	0.0408	0.1211
			1000	0.000786	0.2444

Table 2: Timings for Multi-Threaded Program on various threads

6.2.1 Analysis

Effect of Thread Count:

- As the number of threads increases, the execution time (both average and MIN_MAX) generally decreases for each dataset.
- This indicates that the program benefits from parallel processing.

Optimal Thread Count:

- There is an optimal number of threads for each dataset beyond which the performance gains diminish.
- For example, in the `data_tiny` dataset, the execution time continues to decrease until around 25 threads, after which it starts to increase.

Dataset Impact:

- The impact of multi-threading is more significant for larger datasets.
- Larger datasets (`data_medium` and `data_large`) show more improvement with an increase in the number of threads compared to smaller datasets.

Observations:

- The `data_tiny` dataset does not benefit significantly from multi-threading, as the overhead of thread management might outweigh the gains.
- For the `data_large` dataset, there's a substantial reduction in execution time with an increase in the number of threads.

7 Overall Analysis:

The performance gains from multi-threading are evident, particularly for larger datasets. However, it is crucial to note that there exists an optimal number of threads, beyond which the overhead of managing additional threads offsets the performance gains. The impact of multi-threading is more pronounced in larger datasets, where parallel processing efficiently distributes the workload. When determining the number of threads, careful consideration of the dataset characteristics and underlying hardware is essential. Additionally, further optimizations, such as load balancing or algorithmic improvements, may be explored to enhance the overall performance of the multi-threaded program. It is important to bear in mind that the effectiveness of these strategies can vary based on the specific nature of the application, the algorithms employed, and the hardware configuration.