Habib University
Operating Systems - CS232

Assignment 04 - Report
Multi-Threading

Instructor: Munzir Zafar

Ali Muhammad Asad - aa07190

Contents

1 Introduction

The assignment aims to use the knowledge of multi-threading using POSIX Threads API to use and implement a multi-threaded-file processor. The goal is to develop a multi-threaded C program that processes a large dataset concurrently. Therefore, synchronization mechanisms will be needed to ensure thread safety and efficient resource utlization

2 Makefile and I/O

2.1 Input

Single Threaded Program

The program takes one additional command line argument; the first one obviously being the name of the program, and the second command line argument will be the name of the data file to be processed. The data file name can also be a path to the directory in which the file is located, or the file can be in the same directory. If the user fails to provide the desired number of arguments, the program will exit with an error message. The program will also exit with an error message if the file provided is not found.

Multi-Threaded Program

The program takes two additional command line arguments; the first one obviously being the name of the program, the second command line argument again being the name of the data file to be processed, and the third command line argument being the number of threads to launch for processing. If the user does not provide the number of threads, then the default value of 4 threads is used for processing. If the user fails to provide the desired number of arguments, the program will exit with an error message. The program will also exit with an error message if the file provided is not found.

2.2 Makefile

The accompanying makefile has also been provided with the program and supports the following

- `make build:` builds the program and generates the executable files titled ‘‘`fps`’’ (file processor single thread) and ‘‘`fpm`’’ (file processor multi-thread)
- `make runs <input_file>:` runs the single threaded program with the given dataset file.
- `make runm <input_file> [num_threads]:` runs the multi-threaded program, with the given dataset file, and provided number of threads.
- `make clean:` cleans the directory by removing the executable files

3 Implementation

3.1 Data Structres

Single Threaded Program

In the single threaded program, the primary data structure being used is a dynamically allocated array ‘‘data’’ that stores the integer data of the file.

```
typedef long long int ll;
ll *data; ll cap = 10;
data = (ll *)malloc(cap * sizeof(ll));
/* Code ... */
  cap *= 2;
  data = (ll*)realloc(data, cap * sizeof(ll));
  /* More Code */
```

Listing 1: Dynamic Data Array

As we are parsing over a huge data set (maybe even more than the sample), the data array is of type “`long long int`” to prevent integer overflows. Additionally, a `cap` was used that is updated if the dataset reaches the capacity of the array, in which case, the cap increases and the array is reallocated to double of its size. This is a common strategy that balances the time spent between frequent allocations and memory overhead.

Multi-Threaded Program

1. **Data Array:** The data array in the multi-threaded prgram remains the same as it was in the single threaded program, with the same logic for dynamic re-allocation.
2. **Thread Data Struct:** A thread data struct was created that would point to the data set, and marks the starting and ending points of the data set that the thread would be processing. Therefore, it encapsulates the data each thread performs its computation on.

```
typedef struct{
  ll* data;
  ll start;
  ll end;
} threadData;
```

Listing 2: Thread Data Struct

3. **Thread Arrays:** Two distinct arrays were used; one that would store the thread identifiers of each worker thread, and one that would keep track of the thread data struct to pass individual thread-specific data.

```
pthread_t threads[num_threads];
threadData thread_data[num_threads];
```

Listing 3: Thread Arrays

The `thread_data` is initialized with the data array, and the starting and ending points of the data set that each thread would be processing. The `threads` array is initialized with the thread identifiers of each thread. They work in tandem to ensure that each thread has

the data it needs to process, and the thread identifier to be able to join the thread later on.

4. **Mutex:** Lastly, two "mutex" of type `pthread_mutex_t`; `mutex_sum` and `mutex_minmax` were used to provide a mechanism for synchronizing access to shared global variables. This would ensure that updates to the global variables are thread-safe, preventing rac conditions.

3.2 Algorithms and Code

3.2.1 Single Threaded Program

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

#define ITERS 100
typedef long long int ll;

int main(int argc, char *argv[]){
  if (argc != 2){ // Check if the number of arguments is correct
    printf("Usage: %s <input_file>\n", argv[0]); return 1;
  }
  clock_t start = clock(); // Start timer for reading file
  FILE *datafile = fopen(argv[1], "r"); // Open file in read mode
  if(datafile == NULL){ // Check if file exists
    fprintf(stderr, "Error: Could not open file %s. Please check if the file with the given name exists\n", argv[1]); return 1;
  }

  // Allocate memory for data
  ll *data, *temp; ll size = 0, cap = 1000;
  data = (ll *)malloc(cap * sizeof(ll));
  if(data == NULL){ // Check if memory allocation was successful
    fprintf(stderr, "Error: Memory allocation failed!\n");
    fclose(datafile); return 1;
  }

  // While there is data in the file, add it to the integer array
  while(fscanf(datafile, "%lld", &data[size]) == 1){
    size++; // Increment size
    if(size == cap){ // If size is equal to capacity, double the capacity and dynamically reallocate memory
      cap *= 10;
      temp = (ll *)realloc(data, cap * sizeof(ll));
      if(temp == NULL){ // Check if memory reallocation was successful
        fprintf(stderr, "Error: Memory allocation failed!\n");
        free(data); fclose(datafile); return 1;
      } data = temp; // If successful, assign the new memory location to data
    }
  } fclose(datafile);
  // End timer for reading file and print the time taken
  clock_t end = clock();
```

```
41    double elapsed_time = (double)(end - start) / CLOCKS_PER_SEC;
42    printf("Time taken to read file: %lf\n", elapsed_time);
43
44    // Calculating sum, min and max values of the data set
45    double sum_time = 0, minmax_time = 0; ll sum, min, max;
46    // Run the function ITERS times and calculate the average time taken
        for sum, and, minimum and maximum
47    for(size_t j = 0; j < ITERS; j++){
48      clock_t start_sum = clock(); sum = 0;
49      for(int i = 0; i < size; i++) sum += data[i];
50      clock_t end_sum = clock();
51      sum_time += (double)(end_sum - start_sum) / CLOCKS_PER_SEC;
52    }
53    for(int j = 0; j < ITERS; j++){
54      clock_t start_minmax = clock();
55      min = LLONG_MAX; max = LLONG_MIN;
56      for(int i = 0; i < size; i++){
57        if(data[i] > max) max = data[i];
58        if(data[i] < min) min = data[i];
59      }
60      clock_t end_minmax = clock();
61      minmax_time += (double)(end_minmax - start_minmax) / CLOCKS_PER_SEC;
62    }
63
64    printf("Sum: %lld \t Min: %lld \t Max: %lld\n", sum, min, max);
65    printf("Average time taken to calculate sum: %lf\n", sum_time / ITERS);
66    printf("Average time taken to calculate min and max: %lf\n",
        minmax_time / ITERS);
67
68    // Free the memory allocated to data
69    free(data);
70    return 0;
71  }
```

Listing 4: Single Threaded Program label

The listing above shows the single threaded program code. It starts by checking if the suer has provided the correct number of arguments, and its correctness. It then proceeds to open the file provided by the user, reading it line by line, and storing the data in the dynamically allocated array. Once the file has been read, the program proceeds to parse over the data set, first computing the sum of the data, and calculating the average time taken to compute the sum. It then proceeds to compute the minimum and maximum of the data set, and calculates the average time taken to compute the minimum and maximum. The average time taken is calculated by running the same computation 100 times, and taking the average of the time taken across these 100 runs. The program then frees the dynamically allocated array, and exits.

3.2.2 Multi-Threaded Program

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

#define ITERS 100
#define DEFAULT_NUM_THREADS 4

typedef long long int ll;

// Global Shared Variables
ll global_sum = 0, global_min = LLONG_MAX, global_max = LLONG_MIN;

// Mutex for synchronization
pthread_mutex_t mutex_sum, mutex_minmax;

// Thread Data Structure
typedef struct{
  ll* data;           // Pointer to the data array
  ll start;           // Start index of the chunk
  ll end;             // End index of the chunk
} threadData;

void* compSum(void* arg){
  threadData* tdata = (threadData*) arg;
  ll* data = tdata->data; // Pointer to the data array
  ll sum = 0;
  for(size_t i = tdata->start; i < tdata->end; i++) sum += data[i];
  // Update global variables
  pthread_mutex_lock(&mutex_sum);
  global_sum += sum;
  pthread_mutex_unlock(&mutex_sum);
  pthread_exit(NULL);
}

void* compMinMax(void* arg){
  threadData* tdata = (threadData*) arg;
  ll* data = tdata->data; // Pointer to the data array
  ll min = LLONG_MAX, max = LLONG_MIN;
  for(size_t i = tdata->start; i < tdata->end; i++){
    if(data[i] > max) max = data[i];
    if(data[i] < min) min = data[i];
  }
  // Update global variables
  pthread_mutex_lock(&mutex_minmax);
  if(min < global_min) global_min = min;
  if(max > global_max) global_max = max;
  pthread_mutex_unlock(&mutex_minmax);
  pthread_exit(NULL);
}

```

```c
int main(int argc, char* argv[]){
  // Check if the number of arguments is correct
  if(argc < 2 || argc > 3){
    fprintf(stderr, "Usage: %s <input_file> [num_threads] where num_threads is optional, if not provided, the default is 4\n", argv[0]); return 1;
  }
  int num_threads = DEFAULT_NUM_THREADS; // Default number of threads
  if(argc == 3){ // If number of threads is provided, check if it is valid
    num_threads = atoi(argv[2]);
    if(num_threads < 1){
      fprintf(stderr, "Error: Invalid number of threads\n"); return 1;
    }
  }
  clock_t file_start = clock(); // Start timer for reading file
  FILE* datafile = fopen(argv[1], "r");
  if(datafile == NULL){ // Check if file exists
    fprintf(stderr, "Error: Could not open file %s. Please check if the file with the given name exists\n", argv[1]);
    return 1;
  }

  // Allocate memory for data
  ll* data, *temp; ll size = 0, cap = 100;
  data = (ll*)malloc(cap * sizeof(ll));
  if(data == NULL){ // Check if memory allocation was successful
    fprintf(stderr, "Error: Memory allocation failed!\n");
    fclose(datafile); return 1;
  }

  // While there is data in the file, add it to the integer array
  while(fscanf(datafile, "%lld", &data[size]) == 1){
    size++; // Increment size
    if(size == cap){ // If size is equal to capacity, double the capacity and dynamically reallocate memory
      cap *= 10;
      temp = (ll*)realloc(data, cap * sizeof(ll));
      if(temp == NULL){ // Check if memory reallocation was successful
        fprintf(stderr, "Error: Memory allocation failed!\n");
        free(data); fclose(datafile); return 1;
      } data = temp; // If successful, assign the new memory location to data
    }
  } fclose(datafile);
  // End timer for reading file and print the time taken
  clock_t file_end = clock();
  double file_read = (double)(file_end - file_start) / CLOCKS_PER_SEC;
  printf("Time taken to read file: %lf\n", file_read);

  // Initialize mutexes
  pthread_mutex_init(&mutex_sum, NULL); pthread_mutex_init(&mutex_minmax, NULL);
```

```
99  double sum_time = 0, minmax_time = 0; // Time taken to calculate sum, min and max
100 // Run the function ITERS times and calculate the average time taken for sum, and, minimum and maximum
101 pthread_t threads[num_threads]; // Array of threads
102 threadData thread_data[num_threads]; // Array of thread data
103 ll chunk_size = size / num_threads; // Size of each chunk
104 for(int i = 0; i < num_threads; i++){
105   thread_data[i].data = data;
106   thread_data[i].start = i * chunk_size;
107   thread_data[i].end = (i == num_threads - 1) ? size : (i + 1) * chunk_size;
108 }
109 struct timespec sum_start, sum_end, minmax_start, minmax_end;
110 clock_gettime(CLOCK_MONOTONIC, &sum_start);
111 for(int j = 0; j < ITERS; j++){
112   global_sum = 0;
113   // Create and launch threads
114   for(int i = 0; i < num_threads; i++) pthread_create(&threads[i], NULL, compSum, &thread_data[i]);
115
116   // Wait for all threads to finish
117   for(int i = 0; i < num_threads; i++) pthread_join(threads[i], NULL);
118 }
119 clock_gettime(CLOCK_MONOTONIC, &sum_end);
120 sum_time = (double)(sum_end.tv_sec - sum_start.tv_sec) + (double)(sum_end.tv_nsec - sum_start.tv_nsec) / 1e9;
121
122 clock_gettime(CLOCK_MONOTONIC, &minmax_start);
123 for(int j = 0; j < ITERS; j++){
124   global_min = LLONG_MAX; global_max = LLONG_MIN;
125   // Create and launch threads
126   for(int i = 0; i < num_threads; i++) pthread_create(&threads[i], NULL, compMinMax, &thread_data[i]);
127
128   // Wait for all threads to finish
129   for(int i = 0; i < num_threads; i++) pthread_join(threads[i], NULL);
130 }
131 clock_gettime(CLOCK_MONOTONIC, &minmax_end);
132 minmax_time = (double)(minmax_end.tv_sec - minmax_start.tv_sec) + (double)(minmax_end.tv_nsec - minmax_start.tv_nsec) / 1e9;
133
134 // Print the sum, min and max values, and the average time taken by each thread and the average elapsed time
135 printf("Sum: %lld \t Min: %lld \t Max: %lld\n", global_sum, global_min, global_max);
136 printf("Average time taken to calculate sum: %lf\n", sum_time / ITERS);
137 printf("Average time taken to calculate min and max: %lf\n", minmax_time / ITERS);
138
139 // Destroy the mutex and free the memory allocated to data
140 pthread_mutex_destroy(&mutex_sum); pthread_mutex_destroy(&mutex_minmax);
```

```
141     free(data);
142     return 0;
143 }
```

Listing 5: Multi-Threaded Program label

The above listing shows the multi-threaded program code. After making the necessary checks, and populating the data into the array, the program initalizes two mutexes; one for the sum and one for min/max operation. It then proceeds to initialize the thread data struct and thread identifier arrays according to the number of threads; default or user provided. It then creates a chunk size of the data set that each thread would be processing. Then for each thread, we point its corresponding struct to the data array, and mark its starting and ending point in the array. Then we create each thread, and pass each thread data struct as an argument to the thread function responsible for computing the sum. Then those threads are joined once they've effectively calcualted their local sums, and updated the global sum variable. The start and end time for this computation is noted over 10 iterations of the same computation to get the average time taken to calcualte the sum. The same process is repeated to compute the minimum and maximum; a different function is used for computing minimum and maximum, and the average time across 100 iterations is noted. The program then frees the dynamically allocated array, and exits.

4 Timing Comparison and Analysis

> * *The following timings were taken on a 4-core 8-Threads Intel i7-6700HQ CPU @ 3.50GHz, therefore, the performance may vary on different systems.*

The timings were taken for 4 sample files; `data_tiny` (1000 integers), `data_small` (1000000 integers), `data_medium` (10000000 integers), and `data_large` (100000000 integers). For each of these data files, the single threaded program was run, and the multi-threaded program was run passing various number of threads as command line arguments. The average time for each program was taken across 10 runs.

4.1 Single Threaded Program

The table below summarizes the average elapsed time for each of the data sets on the single threaded program.

Dataset	Average Time - SUM	Average Time - MIN/MAX
data_tiny	0.000015	0.000015
data_small	0.002580	0.002895
data_medium	0.025862	0.026871
data_large	0.258235	0.272356

Table 1: Timings for Single-Threaded Program

4.2 Multi-Threaded Program

The table below summarize the average time taken by a thread, and the average elapsed time of the multi-threaded program, on various number of threads for each data set.

Tiny Data Set (1000 Integers)			Small Data Set (1000000 Integers)		
# Threads	Avg Time / SUM	Avg Time / Min Max	# Threads	Avg Time / SUM	Avg Time / Min Max
4	0.000099	0.000123	4	0.000870	0.000916
8	0.000199	0.000229	8	0.000862	0.000913
10	0.000275	0.000265	10	0.001062	0.001373
50	0.001496	0.001175	50	0.001578	0.001884
100	0.002790	0.002334	100	0.002531	0.002670
Medium Data Set (10000000 Integers)			**Large Data Set (100000000 Integers)**		
# Threads	Avg Time / SUM	Avg Time / Min Max	# Threads	Avg Time / SUM	Avg Time / Min Max
4	0.006776	0.008611	4	0.062903	0.084454
8	0.005290	0.007864	8	0.051086	0.078059
10	0.007409	0.009972	10	0.061836	0.090346
50	0.007984	0.010042	50	0.059325	0.084275
100	0.008746	0.011659	75	0.052698	0.079279
			100	0.054631	0.081116
			500	0.076485	0.106152

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

4.3 Analysis

The average time for sum is, in general, less than that for min/max. The sum operation is a sequential operation with no branching, thus can be easily pipelined. Finding the min/max involves comparison, which introduces branching, potentially causing pipeline stalls in the CPU. Addition is also generally faster than comparison. Also, accessing data sequentially can be cache friendly; for minimum and maximum, there is an additional overhead of comparison, thus increasing the time taken to compute the minimum and maximum.

Single vs Multi-Threaded

For the tiny data set, the single threaded program performs better than the multi-threaded program. A reason could be that the overhead of thread creation, context switching, and synchronization outweighs the benefits of concurrent execution, as evident in the time elapsed for processing, which does not improve as the number of threads increase.

As the data set size increases, the single-threaded performance degrades, which is expected because the amount of computation increases with the size of the data. The multi-threaded, in general, shows a significant improvement in processing times. This suggests that the computational workload is enough to benefit from parallelization from multiple threads. The performance improvement also suggests good scalability, reflecting efficient parallelization; the multiple threads each get a portion of the data set available to operate on. Owing to concurrency, the time taken to compute the sum, and the minimum and maximum of the entire data set reduces significantly.

Multi-Threaded Performance in terms of Number of Threads

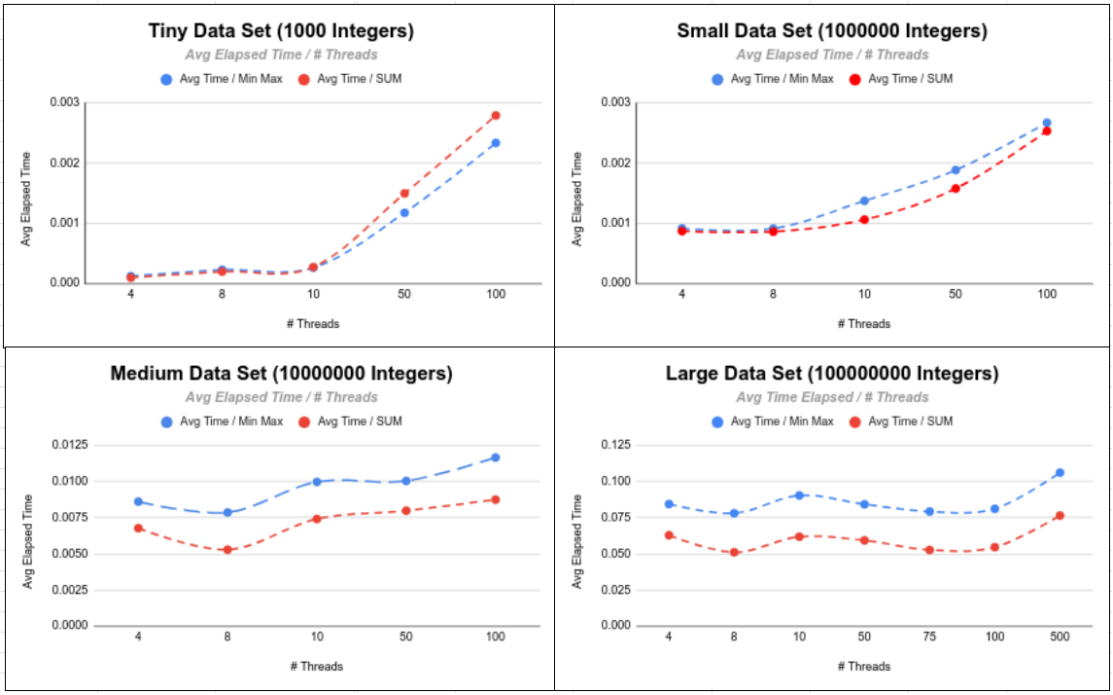

Figure 1: Elapsed Time per # of threads for varying data sets

The figure above summarizes the results of the multi-threaded table above. For the tiny data set, the average elapsed time remains somewhat same up to 10 threads, after which there is a marked increase. This suggests that the overhead associated with thread management — such as creation, context switching, and synchronization — supercedes the execution time for computation of a such a tiny data set. For the small data set, the same can be said, as we see a similar increasing trend in the average elapsed time after 8 threads. As the data set size increases, the benefits of multi-threading become more pronounced (apart from the fact that they outperform single threaded program by a significant amount) as we see a smaller increase (albiet an increase) in between average elapsed times as number of threads increases. This also suggests that there is an optimal thread count that aligns with the system'c core capabilities after which the performance degrades.

**My system has 4 cores with hyperthreading due to which we get 8 threads - 2 threads per core.*

The medium and large data sets exhibit the best performance on 8 threads as compared to 4 threads, likely due to the large computational workload which can effectively utilize the available logical threads provided by the 4 physical cores with hyper-threading (thus 8 threads) on this system. Beyond 8 threads we again see an increase in the average elapsed time, indicating that the performance is again diminshing with increasing threads mainly due to the system's capabilities as the cost of additional threads outweighs the computational benefit as more threads have to be managed by the cpu. This also underlines the importance of balancing thread count with actual core availability - and increasing threads doesn't necessarily imply an increase in overall performance.

In the large data set, however, we see a performance boost after 50 threads, and roughly upto 100 threads after which the performance starts to degrade again. While this does seem counter intuitive, this could be happening due to a variety of reasons for this specific number and on this specific data set. There might be a better cache utlization for certain operations, thus offsetting the overhead of context switching. The underlying thread library and OS scheduler may also have some optimizations that are triggered at this specific number of threads, that the large data set is benefitting from but not the others. The specific way the workload is distributed amongst threads can be a reason for better performance. It could also be due to hyperthreading; for such a large data set, if the threads are waiting on memory, which is common for large data sets, the CPU can switch to the other thread, improving overall throughput. Nonetheless, this is an interesting observation, and I would like to investigate this further.

Overall, the data, as shown in the figure above, indicates that 8 threads deliver the best performance on my system which is capable of running 8 threads simultaneously (seems logical).

5 Takeaway and Reflection

This was the easiest assignment as of yet (thankfully), and a fun one. It was interesting to note that a balance must be kept in all things, even in thread count with actual core availability. Multi-threading does indeed lead to a better performance, but only to a certain extent. So there is an optimal thread count, closely tied to the system's hardware configuration.

6 References

[1] *OpenAI.* ChatGPT [Online]. Available: https://chat.openai.com/ mainly for commenting, and error resolving