



Faculty of Engineering & Technology
Department of Electrical & Computer Engineering
ENCS3390: Operating System Concepts
Second Semester, 2023/2024

Project 1 Report

Student Name: Masa Ahmad Jalamneh
Student ID: 1212145

Section: 1
Instructor: Abdel Salam Sayyad

Date: May-4-2024

1. Abstract:

This project discusses the performance of three different approaches for calculating the average Body Mass Index (BMI) of a given dataset (file). The approaches include a naive implementation, a multiprocessing approach creating multiple child processes, and a multithreading approach creating joinable threads.

The goal is to evaluate the execution time of each approach and compare their performances.

The project involves:

- implementing the BMI calculation function and testing it on a dataset.
- Each approach to executes and find the BMI average in 3 different approaches.
- In each of the approaches we should measure the execution time.
- At the end comment on the differences in performance, and conclusion.

Table of Contents

1. Abstract:	2
Table of Contents	3
Table of figures	4
List of tables	5
2. Introduction	6
3. Procedure	7
3.1 Naive approach:	7
3.1.1 Code:	7
3.1.2 Discussion:	8
3.2 Multiprocessing approach:	9
3.2.1 Code:	9
3.2.2 Discussion:	10
3.3 Multithreading approach (joinable threads):	11
3.3.1 Code:	11
3.3.2 Discussion:	12
4. Measurements, discussion and conclusion	13
5. Amdahl's law analysis	14
5.1 Multiprocessing:	14
5.4 Multithreading:	15

Table of figures

Figure 1: Naive function code.....	7
Figure 2: BMI calculation.....	7
Figure 3: naive execution time and result	8
Figure 4: pipes creation.....	9
Figure 5: child processes in main.....	9
Figure 6: child function.....	9
Figure 7: parent process and avg bmi	10
Figure 8: Joinable threads	11
Figure 9: Thread function	11
Figure 10: multiprocessing whole code execution time	14
Figure 11: multiprocessing parallel oart execution time.....	14
Figure 12: multithreading whole code execution time	15
Figure 13: multithreading parallel part execution time.....	15

List of tables

Table 1: multiprocessors	10
Table 2: multithreading	12
Table 3: results	13

2. Introduction

Naive approach:

‘The naive approach involves a basic, straightforward implementation of the BMI calculation without utilizing any concurrent processing techniques to optimize performance’.

It is simple to implement, it may not be the most efficient method, especially for large datasets. Each operation waits for the previous one to finish before proceeding to the next.

Multiprocessing:

‘The multiprocessing approach is basically utilizing multiple concurrent processes to perform computations simultaneously’.

The multiprocessing approach distributes the workload among multiple child processes, by dividing the dataset into smaller chunks, each chunk is assigned to a child process to be executed.

The multiprocessing approach compared to a naive sequential approach offers improvements in performance, especially when dealing with large datasets; by dividing the work into multiple processes.

Multithreading:

‘Multithreading is a programming technique that allows a single process to execute multiple threads concurrently within the same address space. Each thread represents an independent flow of execution within the process, capable of performing tasks simultaneously’.

‘And joinable threads: when all threads have completed their tasks, they can be joined back to the main thread, allowing the program to consolidate the results or perform any final processing steps’.

3. Procedure

3.1 Naive approach:

3.1.1 Code:

Naive function code:

```
13  //////////////////////////////////////
14  float naive(float h, float w)
15  {
16      float bmi;
17      float height_in_meter= h / 100.0;
18
19      bmi = w /(height_in_meter*height_in_meter);
20
21      return bmi;
22  }
23
24  //////////////////////////////////////
```

Figure 1: Naive function code

File reading and BMI calculations (and final Average value):

```
53  int c1=0;
54  char l[max_length_lines];
55
56
57  while (fgets(l, sizeof(l), file) != NULL) // read file and calculate bmi for each line
58  {
59
60
61
62      g = strtok(l, ",");
63      H_string = strtok(NULL, ",");
64      W_string = strtok(NULL, ",");
65
66      h = atof(H_string);
67      w = atof(W_string);
68
69      // Calculate BMI
70      if (h != 0 && w != 0){
71          BMI= naive(h, w); // function call bmi calculation
72          total+= BMI;
73          c1++;
74      }
75
76
77  }
78
79
80
81  avg_bmi= total / c1;
82
83  printf("BMI average is = %.2lf \n", avg_bmi); // find and print the final result (avg)
84
```

Figure 2: BMI calculation

3.1.2 Discussion:

For this approach (Naïve approach), the code shows a straightforward implementation of the BMI calculation and BMI average.

Starting in the main function by creating and initializing all the parameters we need to use, then since the project is basically requiring reading a dataset to perform operations and find the average BMI of them.

So, the next lines are to open the data file to read it line by line, then split each line into needed tokens (Height, Weight and gender), to perform the BMI calculation on them, by calling the 'naïve' function, which is basically perform the equation of the BMI:

$$\text{BMI} = \text{Weight} \div (\text{Height})^2$$

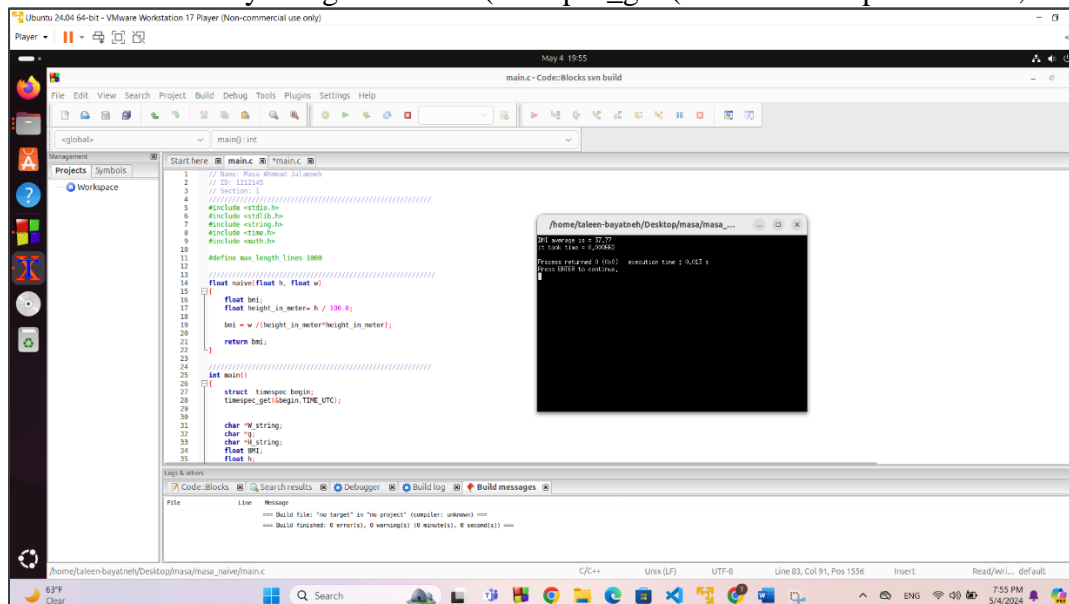
And each line BMI value will be added to the 'total' parameter to find the total BMI values, After the code is reading and calculating the BMI for each line, and incrementing 'c1' which is the counter to count the number of people "lines".

After getting the needed values to calculate the average (total and number of people), the next line will calculate the average and print it.

So, this code I basic and straightforward working in series line by line to give the final result.

The **execution time** this approach took was: **0.000662**

It was calculated by using a function (timespec_get (& struct timespec variable, TIME_UTC)).



```
1 // Name: Hussain Alameen
2 // ID: 1212108
3 // Section: 1
4 // File: main.c
5 #include <stdio.h>
6 #include <math.h>
7 #include <string.h>
8 #include <time.h>
9 #include <unistd.h>
10
11 #define max_length_lines 1000
12
13 // Function to calculate BMI
14 float naive(float h, float w)
15 {
16     float bmi;
17     float height_in_meter = h / 100.0;
18     bmi = w / (height_in_meter * height_in_meter);
19     return bmi;
20 }
21
22 // Main function
23 int main()
24 {
25     // Open the data file
26     FILE *fp = fopen("data.txt", "r");
27     if (fp == NULL)
28     {
29         printf("Error opening file!\n");
30         return 1;
31     }
32     char *W_string;
33     char *H_string;
34     float BMI;
35     float h;
36     float w;
37     float total_bmi = 0.0;
38     int c1 = 0;
39     // Measure execution time
40     struct timespec begin;
41     timespec_get(&begin, TIME_UTC);
42     // Read the data file line by line
43     while (1)
44     {
45         // Read the line
46         char line[max_length_lines];
47         if (!fgets(line, max_length_lines, fp))
48             break;
49         // Split the line into tokens
50         char *tokens[3];
51         int n = sscanf(line, "%f %f %s", &h, &w, tokens);
52         // Calculate BMI
53         BMI = naive(h, w);
54         // Add BMI to total
55         total_bmi += BMI;
56         // Increment counter
57         c1++;
58     }
59     // Calculate average BMI
60     float average_bmi = total_bmi / c1;
61     // Print the result
62     printf("BMI average is : %.2f\n", average_bmi);
63     printf("c1 = %d\n", c1);
64     // Measure execution time
65     struct timespec end;
66     timespec_get(&end, TIME_UTC);
67     // Calculate execution time
68     double execution_time = (end.tv_sec - begin.tv_sec) + (end.tv_nsec - begin.tv_nsec) / 1000000000.0;
69     printf("Execution time : %.6f s\n", execution_time);
70     return 0;
71 }
```

Figure 3: naive execution time and result

Comparison and discussion will be shown later!!

3.2 Multiprocessing approach:

3.2.1 Code:

3.2.1.1 Pipe creation:

```
113     for (int k =0; k<num_c; k++) // loop to create pipes for processes (for communication)
114     {
115         if (pipe(fd[k]) == -1)
116         {
117             printf("error\n");
118         }
119     }
120 }
```

Figure 4: pipes creation

Starting main function with creating parameters I need, then calling the read function to read and store data in a struct, then to implement the multiprocessing I implemented the pipes creation loop to create pipes for the processes I want to have in the code 'num_c'.

3.2.1.2 Child process:

```
127     for (int x = 1; x <= num_c; x++) // loop to create processes
128     {
129         int pid = fork();
130         if (pid == -1)
131         {
132             printf("Fork failed\n");
133         }
134         else if (pid == 0) // child process
135         {
136
137             total += child_process(x,lines_for_c );
138             write(fd[x-1][1], &total, sizeof(float)); //write (pipe)
139             close(fd[x-1][1]);
140             exit(0);
141         }
142     }
```

Figure 5: child processes in main

And after creating the pipes I started loop to create the child (processes), when pid == 0 means that it is the child so call the child function and give it its number and the number of line it must take, and create the communication between child and parent through (read and write) subtotal.

```
72 ///////////////////////////////////////////////////
73 float child_process(int num_c ,double lines_for_c)
74 {
75     int x = lines_for_c*(num_c-1) ;
76     float total =0;
77     for(int j=x+1; j<=lines_for_c+x; j++)
78     {
79         if (j>=lines)
80             break;
81
82
83         total += (bmi_calculator(one_person[j].h, one_person[j].w));
84     }
85     return total;
86 }
87
88 }
```

Figure 6: child function

This is the child function where the child process will find a result for a portion of the whole data, by going through all the provided line for it to find the BMI and return a subtotal to its parent.

3.2.1.3 Parent process:

```
142     else// parent process
143     {
144         close(fd[x-1][1]);
145         read(fd[x-1][0], &total, sizeof(float)); //read (pipe)
146         close(fd[x-1][0]);
147     }
148 }
149
150 float BMI_AVG = total / (500); // calculate the BMI average
151 int j =0;
152 //printf("total=%.2f\n", total);
153 if( j==0)
154 {
155     printf("avg bmi = %.2f\n",BMI_AVG);
```

Figure 7: parent process and avg bmi

Parent process will communicate with its child processes through (read and write ‘pipe’), each child will return a sub result and it will be added to the total to finally find the average BMI: Average BMI= total BMI ÷ number of people.

I tried this code on different number of child processes as shown and tested their execution time:

Table 1: multiprocessors

Number of processes	Execution time	Performance
2	0.001611	620.7324
3	0.003694	270.7092
4	0.002352	425.1700
5	0.006286	159.0836

3.2.2 Discussion:

The number of processes directly impacts execution time and overall performance. Using more processes basically results in faster execution. However, excessive processes can lead to overhead, competing for resources and reducing productivity, especially for a small dataset.

In my case, I ran the code on VirtualBox, my computer has 4 CPUs. This explains why running 5 processes took longer than 4 processes. The optimal number of processes was 4, matching the number of CPUs available.

Comparison and discussion will be shown later!!

3.3 Multithreading approach (joinable threads):

3.3.1 Code:

3.3.1.1 joinable threads:

```
124     for (int i=0; i < numOfThds; i++) // loop to create threads
125     {
126         int* thNUM = malloc(sizeof(int));
127         *thNUM = i;
128         pthread_create(&threads[i],NULL,&thread_process,(void *)thNUM);
129     }
130
131
132     for (int m=0; m < numOfThds; m++)
133     {
134         pthread_join(threads[m],NULL);
135     }
```

Figure 8: Joinable threads

This approach requires creating a multithreading process, I created threads in the main function (create, join and used mutex) using for loop for that. And call the thread function for each thread to perform its part of the task.

3.3.1.2 thread function:

```
76  ///////////////////////////////////////////////////
77  void* thread_process (void* num_of_th) // Threads function
78  {
79
80      int n= *(int*)num_of_th; // number of thread (sequential- passed from the for loop in main function)
81      int y= th_lines*(n);
82      float sub_total=0;
83      pthread_mutex_lock(&mutex1);
84
85      for(int j=y+1; j<=th_lines+y ; j++)
86      {
87          if (j>=num_p)
88              break;
89
90          sub_total += (bmi_calculator(one_person[j].h, one_person[j].w));
91      }
92      thread_f ++;
93      total+=sub_total;
94      pthread_mutex_unlock(&mutex1);
95
96
97      if (thread_f==numOfThds)// calculate the BMI average - final result
98      {
99          printf("bmi avg = %.2lf\n", total/ (num_p-1));
100      }
```

Figure 9: Thread function

The thread function is basically do most of the work, by firstly calculate sub total for the thread portion of data, then add the sub total to the final total each time a thread finishes its calculations. And finally after all threads are done the final total is completed and I can now perform the average equation on it and print it.

3.3.1.3 main:

The main function includes creating all parameters and initialize them then call the read function to read and store all data in a struct, also include giving each thread the data portion in addition to function calls.

I tried this code on different number of threads as shown and tested their execution time:

Table 2: multithreading

Number of threads	Execution time	Performance
2	0.001017	983.2841
3	0.001326	754.1478
4	0.001439	694.9270
5	0.001179	848.1764

3.3.2 Discussion:

By comparing different number of threads (2,3,4 and 5 thread), looks like that larger number of threads means less execution time and better performance.

But the differences do not show clearly because the data we are performing on is a small dataset, this can result in differences.

Comparison and discussion will be shown later!!

4. Measurements, discussion and conclusion

The results are displayed in the table below:

Table 3: results

Approach	Execution time	Performance
Naive	0.000662	1510.570
Multiprocessor 2 processor	0.001611	620.7324
Multiprocessor 3 processor	0.003694	270.7092
Multiprocessor 4 processor	0.002352	425.1700
Multiprocessor 5 processor	0.006286	159.0836
Multithreading 2 threads	0.001017	983.2841
Multithreading 3 threads	0.001326	754.1478
Multithreading 4 threads	0.001439	694.9270
Multithreading 5 threads	0.001179	848.1764

Due to the small dataset the actual result does not appear, but in general using multiprocessing and multithreading must result in better performance, because these approaches split the work in chunks and make (children or threads), work in parallel to finish the work needed reducing the execution time needed to finish the job, so basically multiprocessing and multithreading are better when it comes to large dataset.

More over multiple threads sharing the same memory space but operating within a single process. This makes communication easier, but because shared resources are involved, careful synchronization is required. However, multiprocessing uses several processes, each of which has its own memory space.

My analysis showed that using either **2** (optimal) processes or threads worked best for getting the most out of performance. This approach balanced the benefits of running tasks

simultaneously while avoiding the problems that can come with too much happening at once. It highlights the importance of picking the right multiprocessing or multithreading method for the job and the system we are working with.

5. Amdahl's law analysis

5.1 Multiprocessing:

execution to whole code: 0.003111

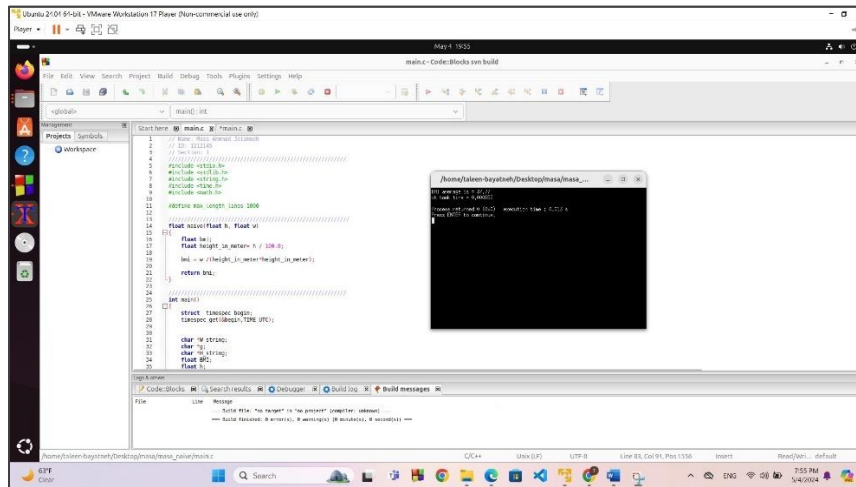


Figure 10: multiprocessing whole code execution time

execution for parallel part: 0.002290

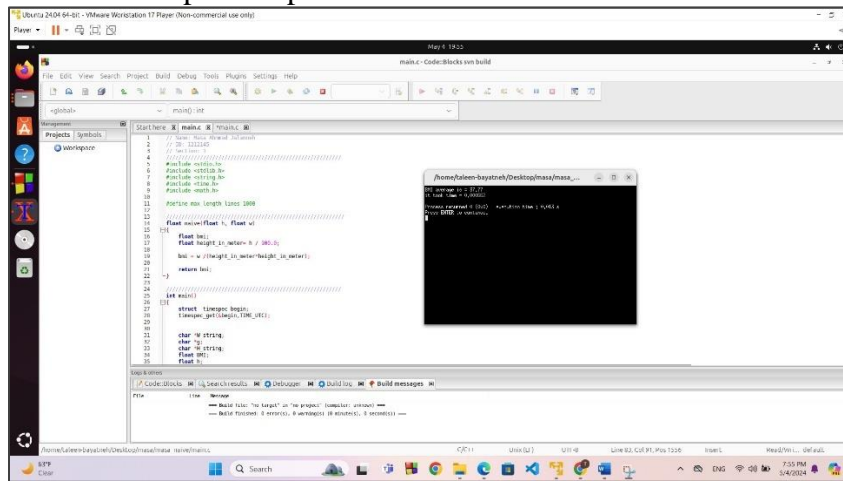


Figure 11: multiprocessing parallel part execution time

N (number of processes) = 4 processes

S (serial portion) = $0.000330 / 0.003111 = 0.106 = 10\%$

P (Parallel portion) = $\text{parallel} / \text{total} = 0.002290 / 0.003111 = 0.73 = 73\%$

$S = 1 - P$ (approximately)

S (speedup) = $1 / ((1 - P) + (P / N)) = 1 / (0.27 + 0.1825) = 2.209 \#$

5.4 Multithreading:

execution for whole code: 0.001852

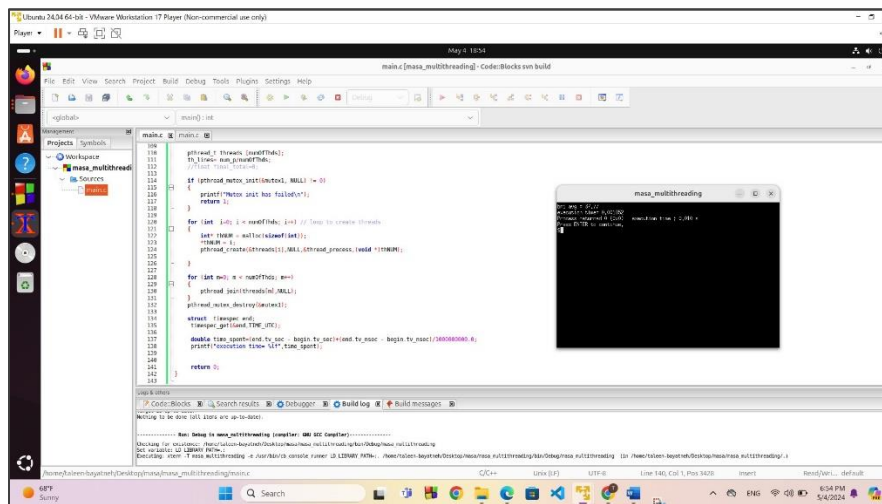


Figure 12: multithreading whole code execution time

execution for parallel part: 0.000824

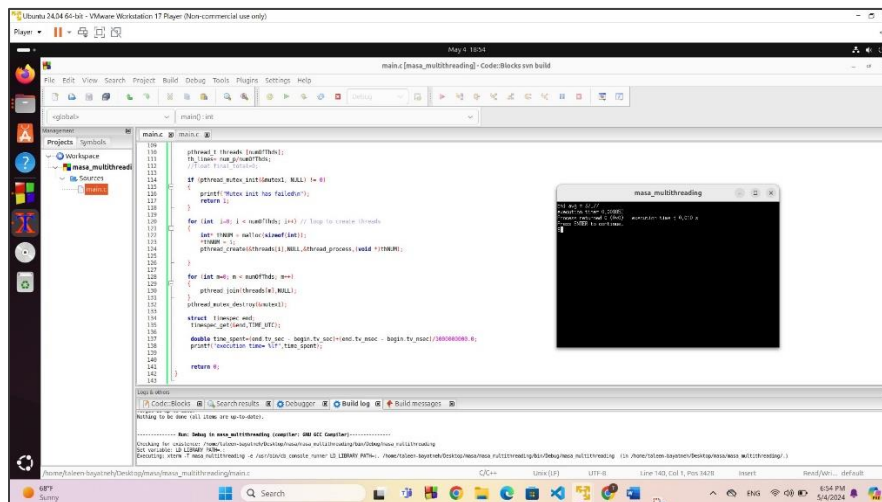


Figure 13: multithreading parallel part execution time

N (number of threads) = 4 threads
 S (serial portion) = $0.000259 / 0.001852 = 0.139 = 14\%$
 P (Parallel portion) = $\text{parallel}/\text{total} = 0.000824 / 0.001852 = 0.44 = 44\%$
 S = 1-P (approximately)
 S (speedup) = $1 / ((1-P) + (P/N)) = 1 / (0.56 + 0.11) = 1.49 \#$