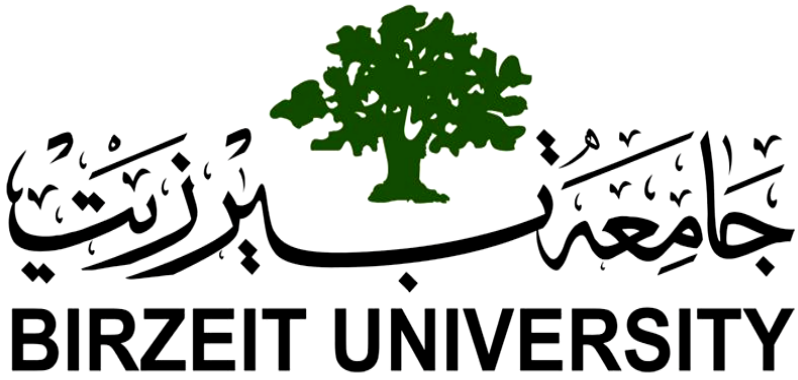

Faculty of Engineering & Technology
Electrical & Computer Engineering Department
Operating Systems
ENCS3390
Report No.1
Project1

Prepared by:

Ahmad Abu Saleem 1201315

Instructor: **Dr. Abdel Salam Sayyad**

Section: 2

Date: 4/5/2024

Abstract

This project involves computing the average Body Mass Index (BMI) using different methods and evaluating their performance. We compare a single-threaded approach (naive), a multithreading method, and a multiprocessing method. In the naive approach, we calculate bmi using one thread, while the multithreading method divides the data into chunks and processes them using multiple threads. The multiprocessing method uses child processes to handle different sections of the data simultaneously. By testing each approach, we examine how they perform and identify their advantages and drawbacks in terms of efficiency and scalability. This helps us understand the best way to manage threads and processes for computing bmi.

Table Of Content

List Of Figures

List Of Tables

1. Theory

1.1 Introduction

Multithreading:

Multithreading is a programming technique that assigns multiple threads to a single process, allowing simultaneous execution of multiple tasks. It leverages multi-core processors, accelerates computation, and optimizes system memory usage, improving responsiveness and efficiency in software applications.

Multiprocessing:

Multiprocessing refers to a system that has more than two central processing units (CPUs). Every additional CPU added to a system increases its speed, power and memory. This allows users to run multiple processes simultaneously. Each CPU may also function independently.

In this project, we will design a program that do average BMI calculation multiplication using 3 different methods:

1- Naïve approach
2- multipleprocesses.
3- multiplethreads.

2-Results and Analysis

2.1 The naive approach.

```c
// Calculate BMI
double calculate_bmi(double height, double weight) {
    return weight / (height * height);
}

// Naive approach to calculate average BMI
double naive_average_bmi(const char *filename) {
    FILE *file = fopen(filename, "r");
    if (file == NULL) {
        printf("Error: Unable to open the file '%s'.\n", filename);
        return -1.0;
    }

    char line[256];
    double total_bmi = 0.0;
    int count = 0;

    // Skip the header line
    fgets(line, sizeof(line), file);

    // Process each line in the file
    while (fgets(line, sizeof(line), file)) {
        char gender[10];
        double height, weight;

        // Parse the line
        if (sscanf(line, "%9[^,],%lf,%lf", gender, &height, &weight) == 3) {
            height = height / 100.0; // Convert height from cm to meters
            double bmi = calculate_bmi(height, weight);
            total_bmi += bmi;
            count++;
        }
    }

    fclose(file);

    // Calculate the average BMI
    return count > 0 ? total_bmi / count : -1.0;
}
```

Figure 1- naive approach code.

The naive approach in the code calculates the average BMI sequentially by reading a CSV file line by line. It computes the BMI for each entry and accumulates the total BMI and count. This straightforward approach does not leverage parallel processing techniques, making it easier to understand but potentially less efficient for large datasets.

Results of naive approach (time in seconds):

RUN 1	RUN 2	RUN 3	RUN 4	RUN 5	Avg
0.000320	0.000316	0.000618	0.000307	0.000319	0.000376

Table 1: Naive approach results

2.2Multiprocess approach

In the mutli processes approach, the task of calculating the average BMI is distributed among multiple child processes created using the fork() function. Each child process calculates the BMI for a portion of the data file

```
    double total_bmi = 0.0;
    int total_count = 0;
    pid_t *pids = malloc(num_processes * sizeof(pid_t));
    int pipes[num_processes][2];  // Create pipes for each child process

    // Create child processes
    for (int i = 0; i < num_processes; i++) {
        // Create pipes
        if (pipe(pipes[i]) == -1) {
            printf("Error: Failed to create pipe %d.\n", i);
            exit(1);
        }
        // Fork a child process
        pid_t pid = fork();
        if (pid < 0) {
            printf("Error: Failed to fork child process %d.\n", i);
            exit(1);
        } else if (pid == 0) {
            // Child process
            close(pipes[i][0]); // Close read end of the pipe
            // Calculate the number of lines to process
            int num_lines = lines_per_process + (i < remaining_lines ? 1 : 0);
            // Calculate the starting line
            int start_line = i * lines_per_process + (i < remaining_lines ? i : remaining_lines);
            // Open the file again in the child process
            FILE *file_child = fopen(filename, "r");
            if (!file_child) {
                printf("Error: Unable to open file '%s' in child process %d.\n", filename, i);
                exit(1);
            }
            // Calculate BMI in chunk
            BMIResult result = process_bmi_chunk(file_child, start_line, num_lines);
            // Close the file
            fclose(file_child);
            // Write the result to the pipe
            write(pipes[i][1], &result, sizeof(BMIResult));
            // Close the write end of the pipe
            close(pipes[i][1]);
            // Exit the child process
            exit(0);
```

Figure 2- part from Multiprocess approach code

In this approach, the parent process creates child processes using fork(), and each child process calculates the BMI for a portion of the data file. The parent process coordinates the child processes, using pipes for communication and synchronization. After all child processes are complete, the parent process reads the results from each child, aggregates them, and calculates the final average BMI.

Results of Multi processes approach (time in seconds):

Processes	Run1	Run2	Run3	Run4	Run5	Avg
2	*0.001082*	*0.001010*	*0.006844*	*0.000898*	*0.001131*	*0.002193*
4	*0.000953*	*0.000836*	*0.000795*	*0.000806*	*0.000716*	*0.000821*

Table 2: multiprocesses approach result

2.3 MultiThreads approach

```c
// Function for thread to process a chunk of the file
void *bmi_thread_worker(void *arg) {
    ThreadArgs *thread_args = (ThreadArgs *)arg;
    const char *filename = thread_args->filename;
    int start_line = thread_args->start_line;
    int num_lines = thread_args->num_lines;

    // Open the file
    FILE *file = fopen(filename, "r");
    if (file == NULL) {
        printf("Error: Unable to open the file '%s'.\n", filename);
        pthread_exit(NULL);
    }
    // Calculate BMI in the specified chunk
    BMIResult result = process_bmi_chunk(file, start_line, num_lines);
    // Close the file
    fclose(file);

    // Return the result as a pointer
    BMIResult *result_ptr = malloc(sizeof(BMIResult));
    if (!result_ptr) {
        printf("Error: Failed to allocate memory.\n");
        pthread_exit(NULL);
    }
    *result_ptr = result;
    // Exit the thread with the result
    pthread_exit(result_ptr);
}

// Function to calculate average BMI using multi-threading
double threaded_average_bmi(const char *filename, int num_threads) {
    pthread_t threads[num_threads];
    ThreadArgs thread_args[num_threads];
    int total_lines = count_lines_in_file(filename);
    int lines_per_thread = total_lines / num_threads;
    int remaining_lines = total_lines % num_threads;

    double total_bmi = 0.0;
    int total_count = 0;

    // Create threads to process chunks of the file
    for (int i = 0; i < num_threads; i++) {
        int start_line = i * lines_per_thread;
        int num_lines = lines_per_thread;
        if (i == num_threads - 1) {
```

Figure 3- part from Multithread approach code

In this approach, the function pthread_create() is used to spawn multiple threads, each processing a chunk of the data file. The main thread waits for all the threads to complete using pthread_join(), gathering results from each thread and calculating the final average BMI.

Results of Mutlithreads approach (time in seconds):

Threads	Run1	Run2	Run3	Run4	Run5	Avg
2	*0.001995*	*0.004026*	*0.003383*	*0.002523*	*0.002116*	*0.002811*
4	*0.001870*	*0.001105*	*0.002257*	*0.002516*	*0.002807*	*0.002111*

Table 3: multi threads approach results

1) How you achieved the multiprocessing and multithreading requirements, i.e. The API and functions that you used?

To meet the requirements for both multiprocessing and multithreading, I utilized various functions and APIs available in the Linux system. These approaches allowed for parallel processing of data, improving the efficiency of my program.

Multithreading

Thread Creation: I used the pthread_create() function to spawn multiple threads that process different portions of the data file simultaneously. The function takes in a thread identifier, optional attributes, a function for the thread to execute, and an argument for that function.

Thread Arguments: I defined a structure (ThreadArgs) to pass specific arguments to each thread. This structure includes the filename, the starting line number, and the number of lines each thread should process.

Thread Execution: Each thread executes a function such as bmi_thread_worker() that calculates BMI for a specific chunk of the file using process_bmi_chunk().

Thread Synchronization and Joining: After creating the threads, I used pthread_join() to wait for each thread to finish its work and retrieve their results for further processing.

Multiprocessing

Process Creation: I used the fork() function to create child processes for parallel data processing. The parent process creates these child processes using a loop, delegating work across them.

Process Arguments: Similar to the multithreading approach, each child process calculates BMI for its assigned portion of the file using process_bmi_chunk().

Inter-process Communication: Pipes (pipe()) facilitated communication between the parent and child processes. Each child writes its results into the pipe, and the parent reads from it to collect the data.

Process Synchronization and Aggregation: Once all child processes were created, the parent process used waitpid() to wait for them to finish. It then read from each child's pipe to aggregate results and calculate the final average BMI.

In both approaches, the primary goal was to divide the workload across multiple threads or processes to optimize file reading and data processing. This parallel processing improved program performance significantly.

Choosing between multiprocessing and multithreading depends on various factors, including memory availability, inter-process communication needs, and the nature of the task. Threads are typically more efficient for CPU-bound tasks, while processes provide better data isolation and are ideal for memory-bound tasks or tasks requiring extensive data sharing.

2) An analysis according to Amdahl's law. What percentage is the serial part of your code? What is the maximum speedup according to the available number of cores? What is the optimal number of child processes or threads?

1) Calculate the Serial Portion

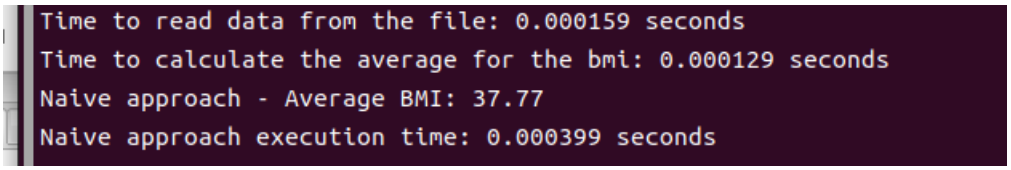

Figure 4- serial portion_time

The serial portion of the code is the sum of the time to read data from the file and the time to calculate the average BMI:

Serial portion=0.000159 s+0.000129 s=0.000288 s

2)the percentage of the serial code

Serial fraction= Serial portion / Naive execution time
Serial fraction=0.0002880/0.000399 = 0.722
The percentage is %72.2

3) the Maximum Speedup Using Amdahl Law

Maximum Speedup = 1 / (0.722 – ((1-0.722)/4))

Maximum Speedup = 1.26

4) Finding the Best Number of Threads or Processes

About 72.2% of the code needs to run one step at a time (serial), so using too many threads or processes won't make things much faster. The maximum possible speedup is around 1.26 times, which means there isn't a lot of room for improvement.

Generally, using around 4 threads or processes (matching the number of available cores) gives you a good balance between speed and complexity.

3) A table that compares the performance of the 3 approaches.

Approach	Avg Execution Time (sec)
Naïve	**0.000376**
Multi Processes (2)	**0.002193**
Multi Processes (4)	**0.000821**
Multi Threads (2)	**0.002811**
Multi Threads (4)	**0.002111**

Table 4: compare the performances

4) Comment on the differences in performance and conclusion.

The naive approach was the quickest, finishing the task in just 0.000376 seconds. This is because it straightforwardly handled everything without using threads or processes, so there was no added overhead.

When we tried using multiple processes, the performance improved. With 2 processes, the average time was 0.002193 seconds. When we increased it to 4 processes, the average time dropped significantly to 0.000821 seconds. This was the fastest of all the approaches, showing that using 4 processes speeds things up without too much extra work.

On the other hand, using multiple threads didn't improve performance as much. With 2 threads, the average time was 0.002811 seconds, and with 4 threads, it was slightly better at 0.002111 seconds. Still, neither option was as fast as the 4-process approach.

Conclusion: The best option is the multi-process approach with 4 processes, which was the fastest and balanced parallelization and overhead well. While using multiple threads did help a bit, it wasn't as effective as using multiple processes.

3)Conclusion

In conclusion, we explored different methods for calculating average BMI using various approaches. The naive approach is straightforward and efficient for small tasks, but it can be slower for larger data sets because it runs sequentially. Using multiple processes can speed things up significantly by splitting the task among several processes. Performance improved with more processes, especially when using 4 processes. multiple threads didn't boost performance as much as multiple processes. However, increasing the number of threads to 4 did provide a small improvement compared to using just 2 threads.

In the end, the most efficient method was the multi-process approach with 4 processes. It offered the fastest execution time and a good balance of parallelization and overhead. This experiment shows the importance of choosing the right method based on the size of the data, the need for synchronization, and the desired efficiency.