Faculty of Engineering and Technology
Electrical and Computer Engineering Department

Operating Systems ENCS3390

First Semester 2024 / 2025

First Project

Student Name: Fadi Bassous **University ID: 1221005**

Instructor’s name: Dr. Mohammed Khalil

Section: 2

Date: 1 /12 /2024

I

Table of contents

Table of Figures

List of Tables:

Theory

1. Naive Approach:

The naive approach executes all tasks sequentially within a single thread, processing one task at a time. This straightforward method does not leverage the advantages of parallelism, resulting in underutilization of multi-core processors. While simple to implement, the lack of concurrency leads to longer execution times, especially for computationally intensive or large-scale tasks.

2. Multiprocessing Approach:

In the multiprocessing approach, multiple independent child processes are spawned to execute tasks in parallel. Each process runs in its own memory space, allowing effective utilization of multi-core CPUs by distributing the workload. This significantly reduces execution time compared to the naive approach. However, creating and managing processes incurs overhead, and inter-process communication can be slower due to isolated memory spaces.

3. Multithreading Approach:

The multithreading approach creates multiple threads within a single process, enabling tasks to run concurrently. Threads share the same memory space, making data sharing between threads more efficient than in multiprocessing. This approach is well-suited for tasks involving high I/O operations or those that can be efficiently parallelized. However, multithreading introduces challenges such as synchronization issues, race conditions, and the potential for deadlocks, which require careful management to ensure correctness and efficiency.

Results:

1. Naive Approach:

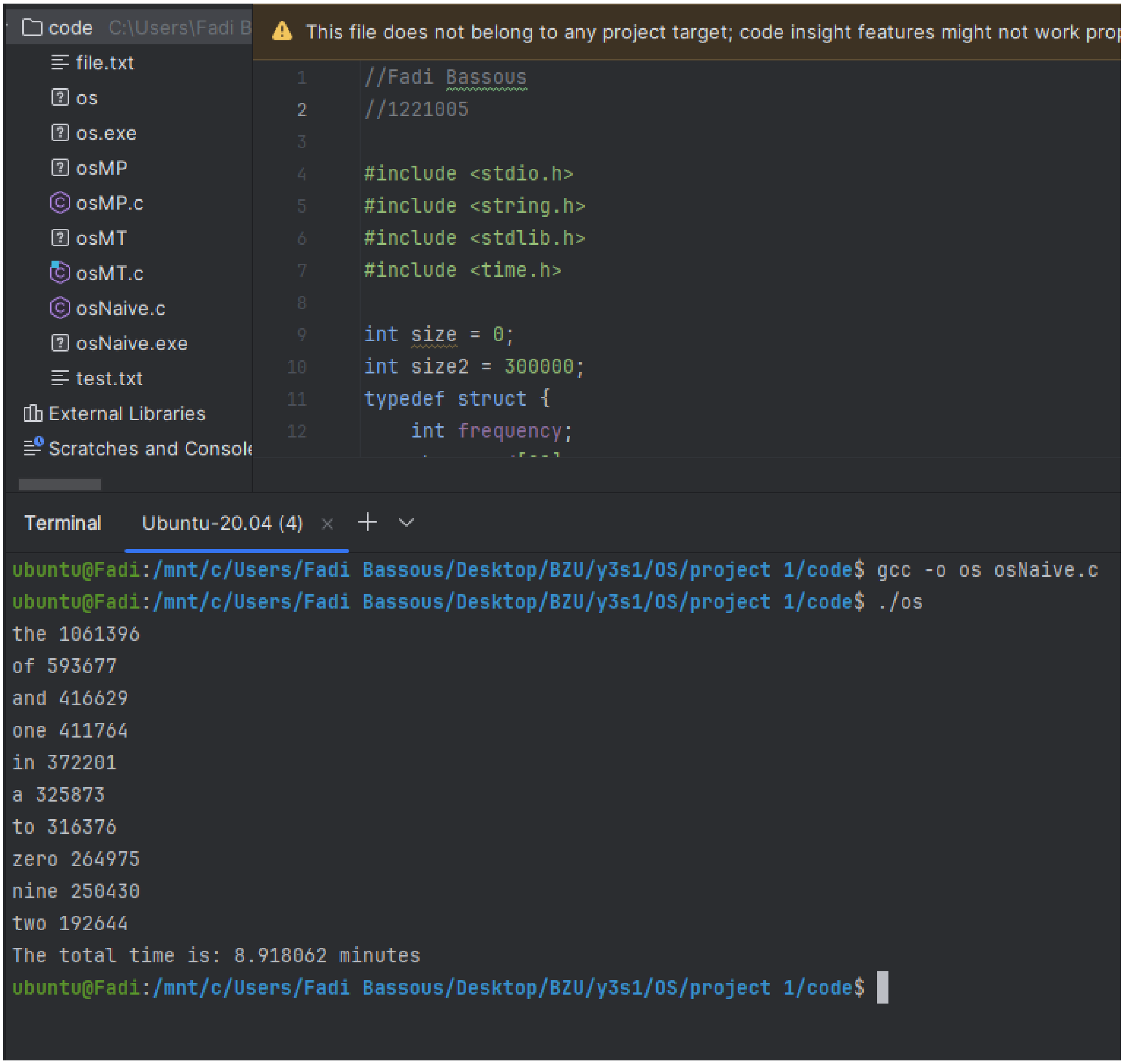

Figure 1: Naive Result

2. Multiprocessing Approach:

8 children:

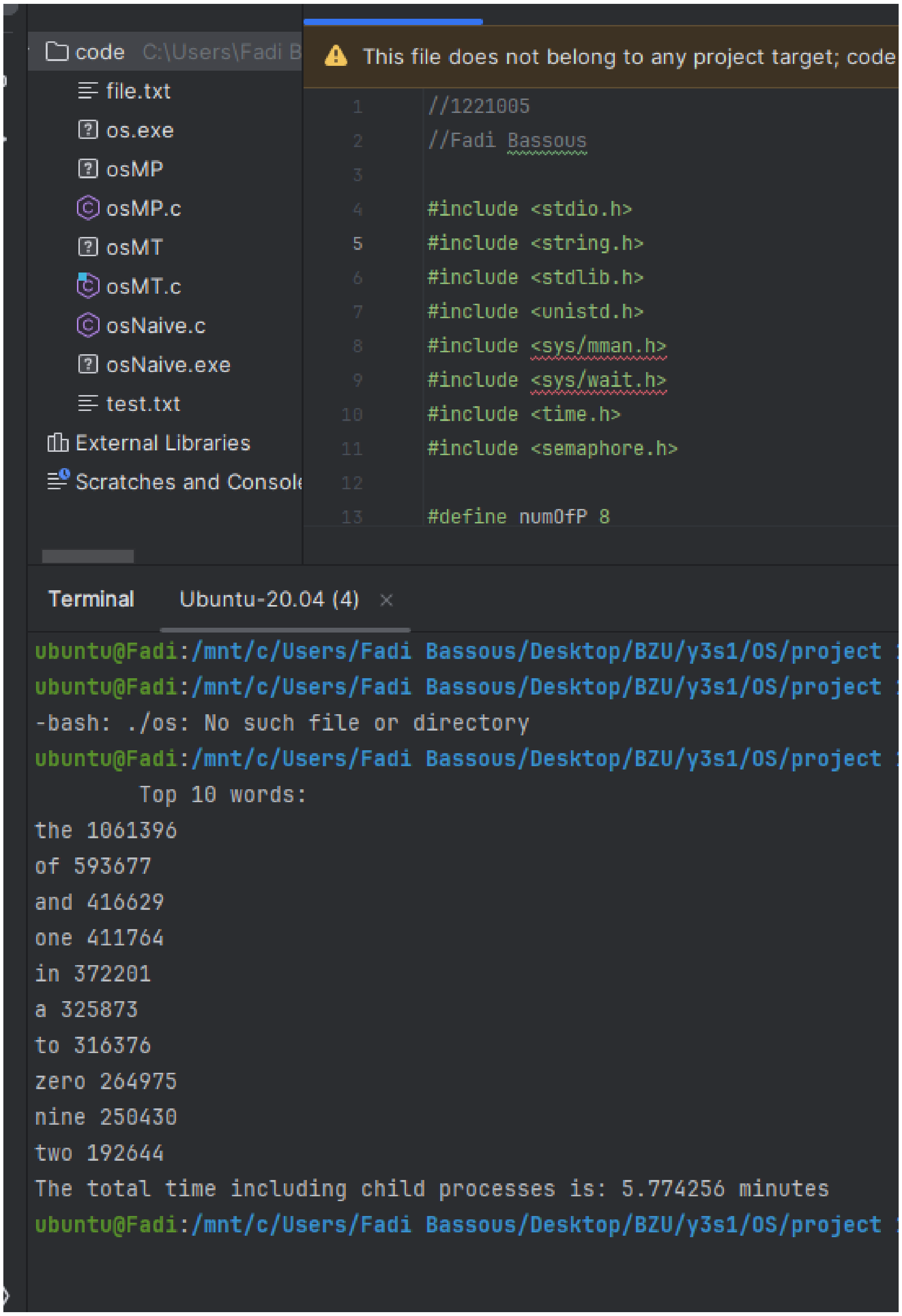

Figure 2: Multiprocessing result (8 children)

6 children:

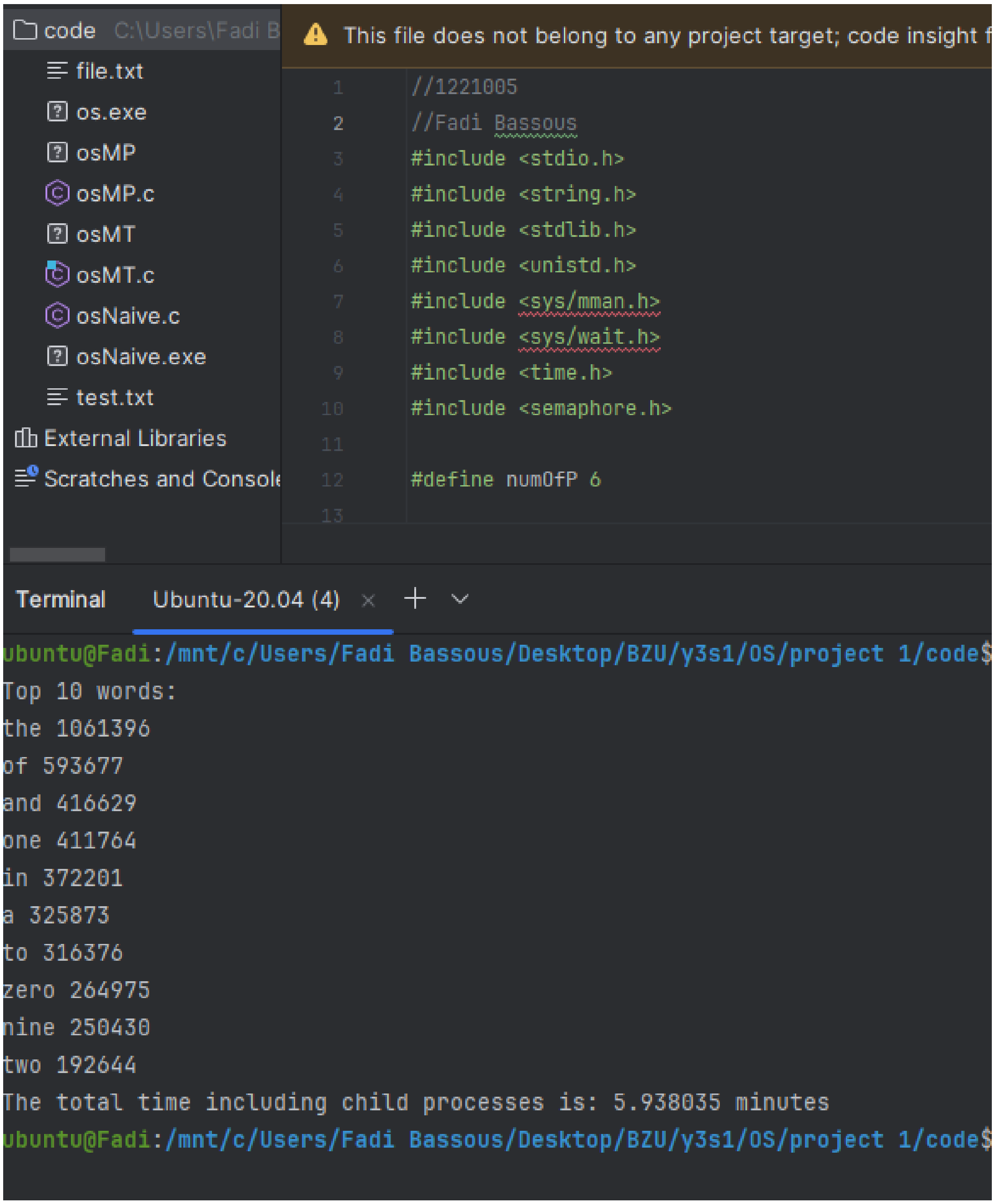

Figure 3: Multiprocessing result (6 children)

4 children:

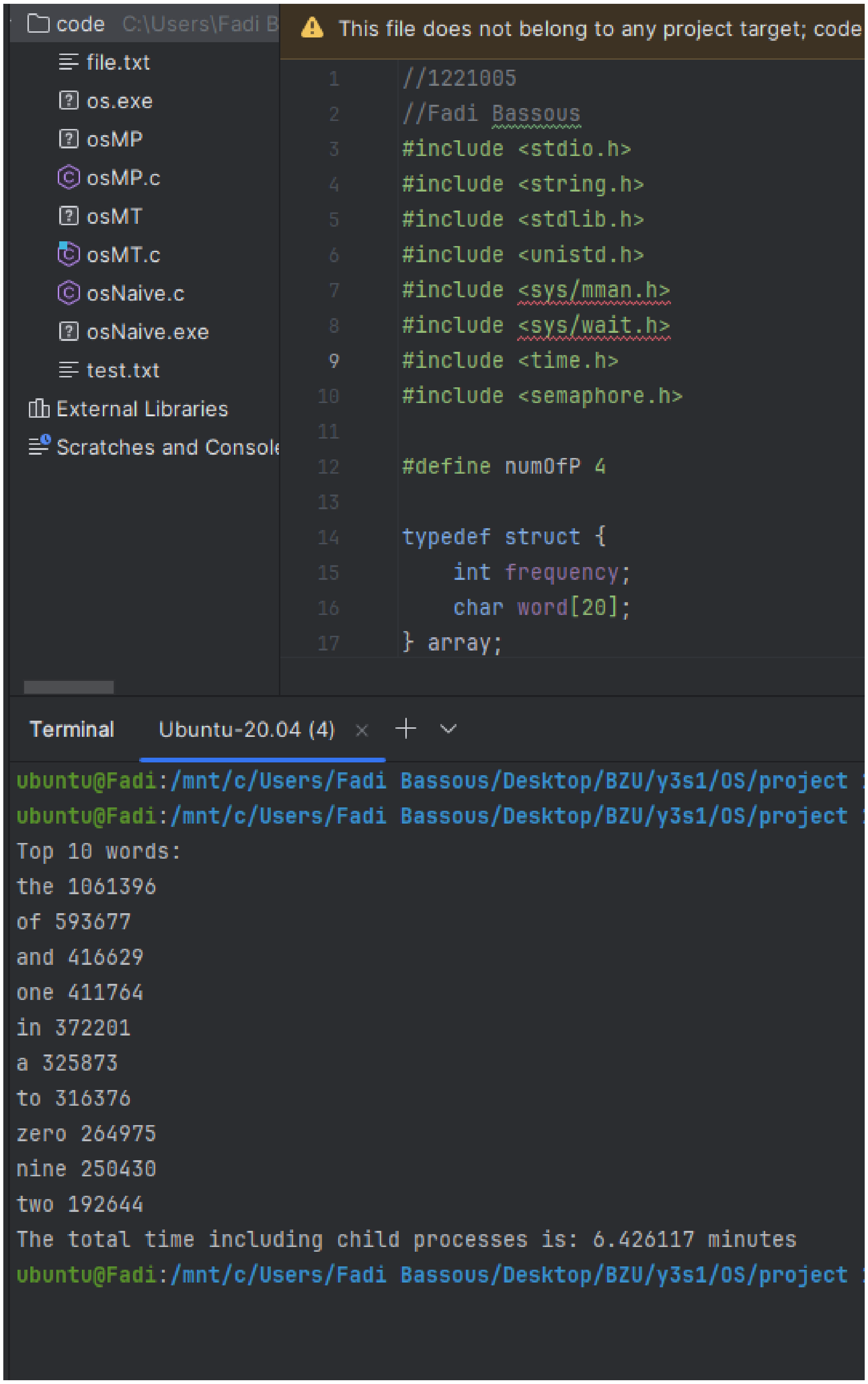

Figure 4: Multiprocessing result (4 children)

2 children:

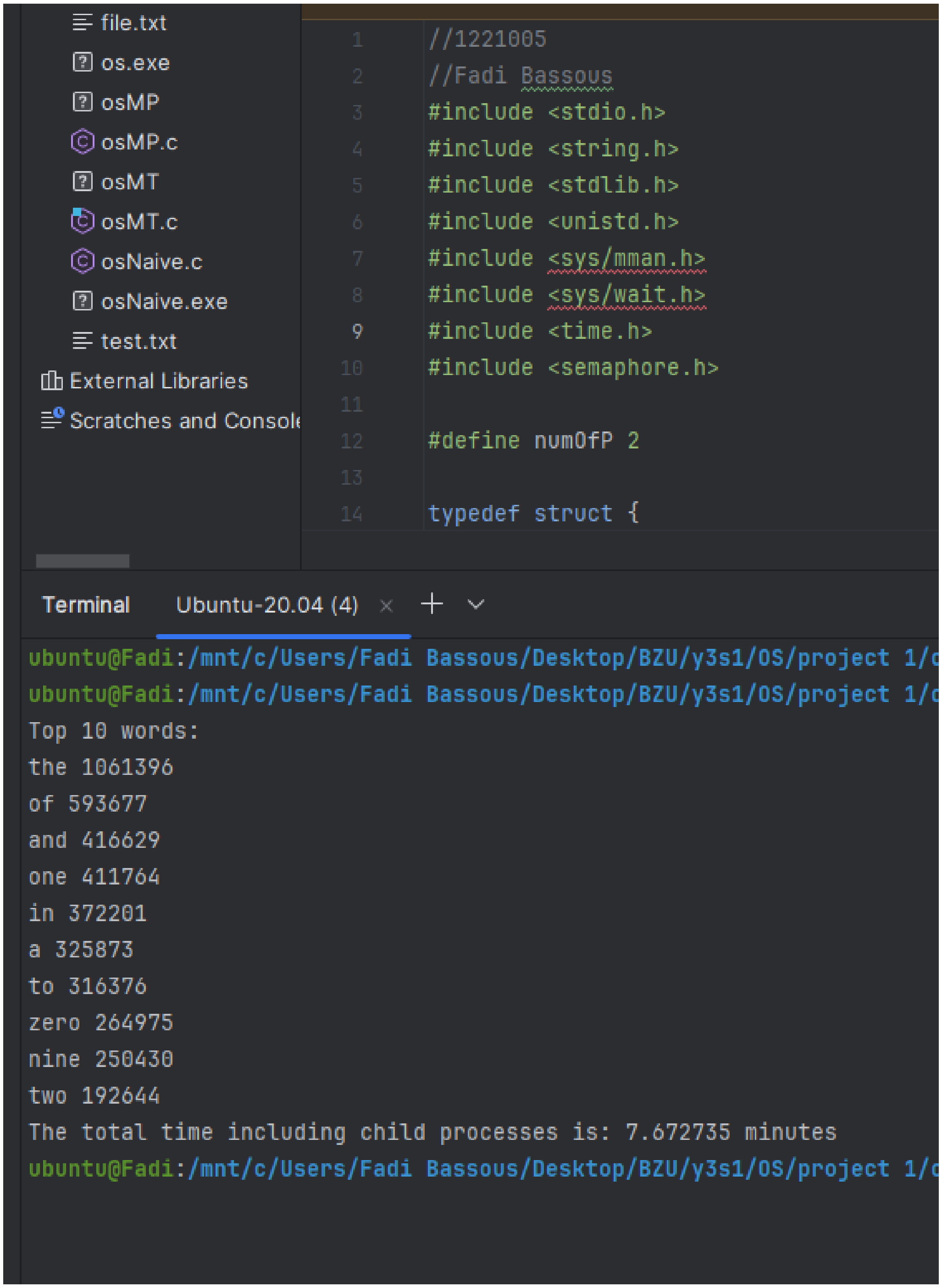

Figure 5: Multiprocessing result (2 children)

3. Multithreading Approach:

8 threads:

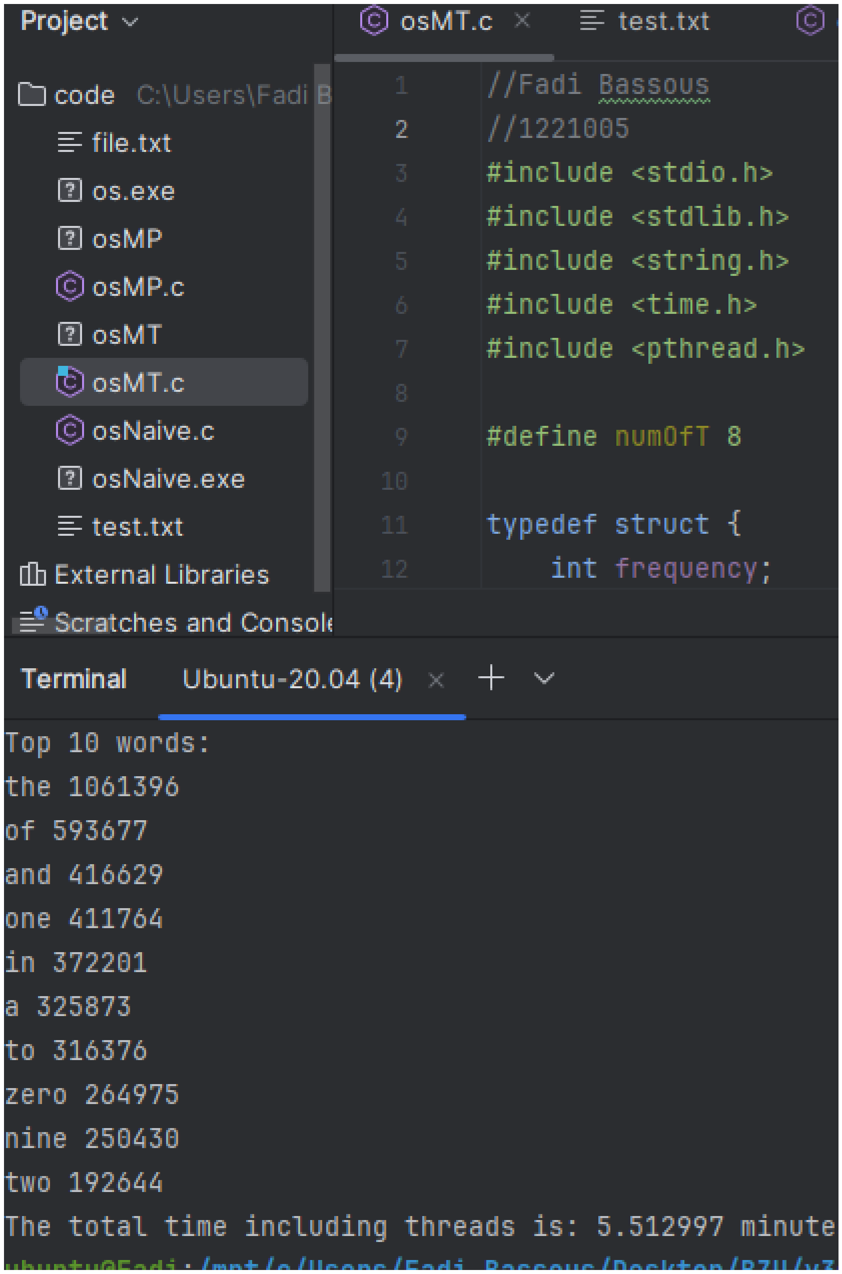

Figure 6: Multithreading result (8 threads)

6 threads:

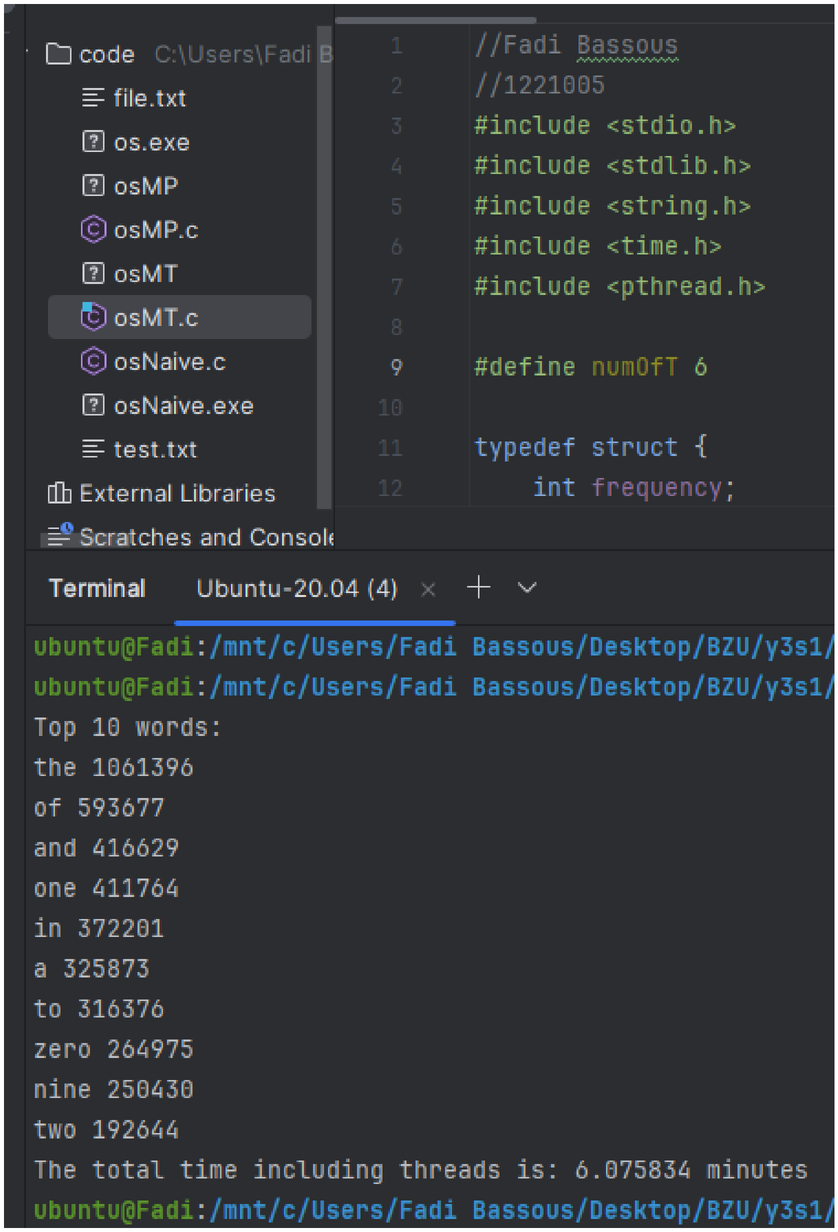

Figure 7: Multithreading result (6 threads)

4 threads:

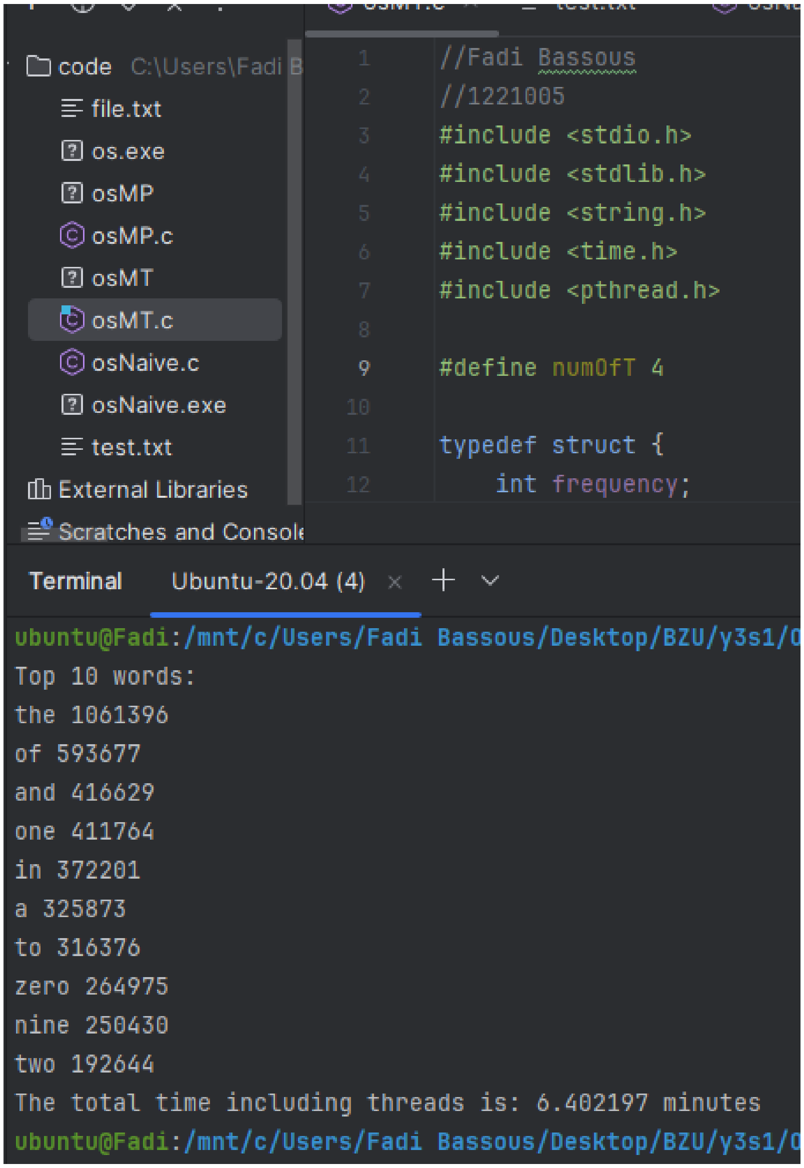

Figure 8: Multithreading result (4 threads)

2 threads:

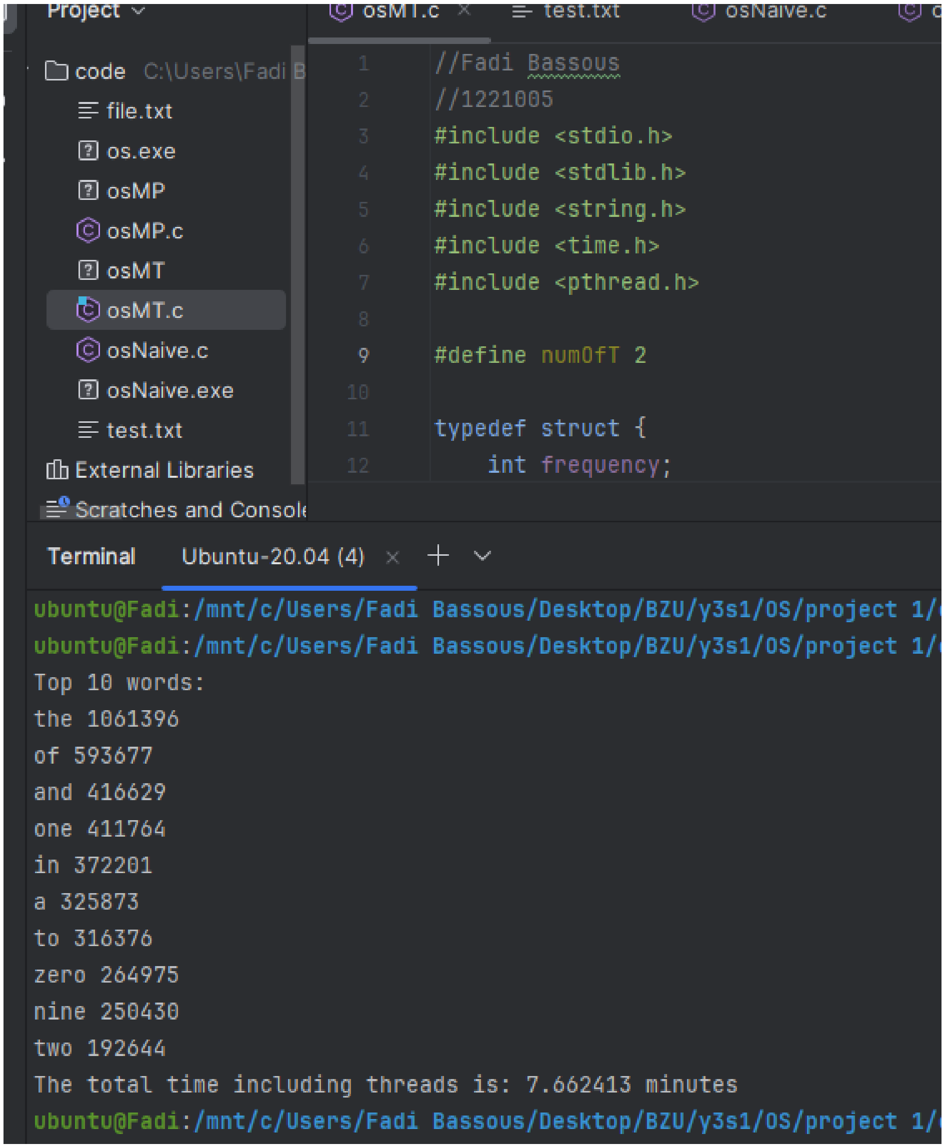

Figure 9: Multithreading result (2 threads)

Questions:

1. Environment Description

- **Computer Specs:**
    - **Processor:** 11th Gen Intel(R) Core(TM) i7-11800H @ 2.30 GHz
    - **Cores:** 8 physical cores, 16 threads (Hyper-threading enabled)
    - **Installed RAM:** 16.0 GB (15.7 GB usable)
    - **Storage:** Local Disk C: 475 GB (90.4 GB free)
    - **System Type:** 64-bit operating system, x64-based processor
- **Operating System:**
    - Windows 11 (configured for WSL Ubuntu Terminal usage)
- **Programming Language:** C
- **IDE Tool:** CLion
- **Virtual Machine:** No virtual machine was used; the code was executed through the Ubuntu terminal configured via WSL (Windows Subsystem for Linux).

2. How You Achieved the Multiprocessing and Multithreading Requirements: The API and Functions Used

- **Multiprocessing**

The following system calls and synchronization mechanisms were utilized to implement multiprocessing:

fork() and **waitpid():**

The fork() system call was used to create multiple child processes, with each process working independently on a portion of the dataset.

The waitpid() function ensured that the parent process waited for all child processes to complete execution before proceeding with result aggregation.

Shared Memory with **mmap()**:

The mmap() system call was used to create shared memory with the MAP_SHARED and MAP_ANONYMOUS flags.

This allowed all child processes to write their computed results into a common memory space, ensuring efficient data sharing.

Semaphores (sem_t):

To prevent data corruption caused by multiple child processes accessing shared memory simultaneously, a semaphore was employed.

The semaphore was initialized using sem_init() and protected critical sections of code where shared memory was updated.

Steps:

Divide the dataset into chunks proportional to the number of child processes (numOfP).

Each child process processes its allocated chunk, calculates word frequencies locally, and stores results in shared memory.

A semaphore ensures that shared memory updates are performed sequentially.

The parent process combines the results after all child processes have completed execution.

- **Multithreading**

The multithreading solution utilized lightweight threads and shared memory for efficient parallelization of computation. Key API functions and synchronization mechanisms used are:

pthread_create() and **pthread_join():**

Threads were created using the pthread_create() function, with each thread working on a designated portion of the dataset.

Threads were joined back to the main thread using pthread_join() upon completion of their tasks, ensuring all computations were complete before merging results.

Mutex (**pthread_mutex_t**):

A mutex was used to ensure thread-safe access to shared resources, such as the result array. This prevented race conditions and maintained data integrity during concurrent updates.

Steps:

Divide the dataset into chunks proportional to the number of threads (numOfT).

Each thread processes its chunk and computes local word frequencies.

A mutex synchronizes access to the shared result array, ensuring orderly updates.

The main thread merges results from all threads after they finish execution.

3. **What percentage is the serial part of your code?**

   Tserial=Ttotal−Tparallel

   Tserial=8.918062−8.361548=0.556514 minutes

   S=Tserial/Ttotal= 0.556514/8.918062≈ 0.0624 (6.24%)

   Calculate the Maximum Speedup?

   Speedup=S+N(1−S)=0.0624+8(1−0.0624)= 1/0.17961≈ 5.57

   his means the maximum speedup with 8 cores is approximately **5.57x**.

4. table that compares the performance of the 3 approaches:

Program / Run time(minute)	Naive	Multiprocessing	Multithreading
___________	8.91	___________	___________
2. Childe/Thread	___________	7.67	7.66
4. Childe/Thread	___________	6.42	6.40
6. Childe/Thread	___________	5.93	6.07
8. Childe/Thread	___________	5.77	5.51

Table 1: Result for all 3 approaches

5. Comment on the differences in performance:

Single-threaded approach:

The single-threaded approach is the most straightforward and, for execution time, the longest. This is because of the inherent lack of parallelism and underutilization of available system resources. It shows the important overhead introduced in transitioning to parallel methods of multiprocessing and multithreading.

Multiprocessing Approach:

Execution time improves considerably while moving from a single process to two child processes, thereby leveraging parallelism effectively. But once the number of child processes is increased beyond this point, the return diminishes. Sometimes the execution time may even stabilize or increase slightly due to the overhead of managing extra processes and inter-process communication.

Multithreading Approach:

With one thread, the execution time is a lot higher compared to the single-threaded and multiprocessing approaches. This is most likely due to the added overhead of managing threads. However, with two threads, the execution time reduces dramatically, making it competitive with the multiprocessing approach. Beyond two threads, additional threads show minimal performance gains; this could be due to synchronization overhead and contention for shared resources.

conclusion

This project demonstrates the significant performance benefits of parallel computing for large dataset analysis. Both multiprocessing and multithreading approaches outperformed the naive method. The study emphasizes the importance of balancing parallelism and overhead for optimal efficiency, guided by theoretical principles such as Amdahl's law.