Faculty of Engineering and Technology
Electrical and Computer Engineering Department

Operating System Concepts
ENCS3390

First Semester, 2024/2025

Project 1

Prepared by:
Qusay Bider **1220649**

Instructor:
AbdelSalam Sayd

Section:4

Abstract

This report examines three different strategies for addressing the challenge of word frequency counting in large text files: single-threaded, multiprocessing, and multithreading. Each method is implemented and evaluated based on performance, resource usage, and scalability.

The single-threaded approach processes the text in a linear fashion, using a simple algorithm, but it encounters considerable performance limitations when handling large datasets. In contrast, the multiprocessing method utilizes multiple processes to distribute the workload, employing shared memory and semaphores for synchronization. Although this approach enhances execution speed by taking advantage of multiple CPU cores, it incurs overhead from process creation and shared memory management. The multithreading method, on the other hand, employs threads to concurrently process different sections of the file, utilizing a global data structure with mutex locks for synchronization. This approach strikes a balance between memory efficiency and high performance, leveraging modern multi-core architectures effectively.

The comparative analysis indicates that while the single-threaded method is straightforward, it is inadequate for large-scale data processing. Both the multiprocessing and multithreading methods significantly improve performance, with multithreading offering the optimal combination of speed and resource efficiency. This study highlights the critical importance of choosing the right parallelization technique for computationally intensive tasks, especially in the realm of large-scale text processing.

In this project we will build:

1) Naive approach

2) Multiprocessing approach

4) Multithreading approach

Environment Description

The development and testing environment for this report included the following:

I. Computer Specifications:

- Processor: 4-core, 8-thread CPU (Intel Core 9300H, 2.40 GHz)
- Memory: 16 GB DDR4 RAM
- Storage: 512 GB SSD

II. Operating System:

- Linux (Ubuntu 20.04 LTS)

III. Programming Language:

- C (with support for POSIX system calls)

IV. Integrated Development Environment (IDE):

- CLion Code

V. Virtual Machine:

- No virtual machine was used; testing was performed on WSL system.

Table of Contents

Table of figures

Implementation Details:

1. Single-threaded Approach

Description: The program processes a file linearly to count word frequencies.

Logic: It reads words one at a time, checks if the word already exists in a data structure, and updates its frequency.

Explain code:

- **Structure Definition:**

  -
    ```
    struct Array {
    char word[MAX_WORD_LENGTH];
    int frequency;
    };
    ```
  - A **word** of up to MAX_WORD_LENGTH characters.
  - Its associated **frequency**, which keeps track of how many times the word appears.

- **Global Variables**
  - **Insert_Array**[MAX_WORDS]: An array to store all unique words and their frequencies, with a maximum size of MAX_WORDS.
  - **size**: Tracks the current number of unique words stored in Insert_Array.

- **Function**
  - **checkIfExist:**
    Checks if a given word already exists in Insert_Array.
  - **loadData:**
    Reads words from a file and populates Insert_Array with unique words and their frequencies.
  - **getTop10:**
    Identifies and prints the top 10 most frequent words in Insert_Array.

Amdahl’s Law Analysis:

$$s = \frac{1}{f + \left(\frac{1 - f}{N}\right)}$$

Where:

- S: Speedup with N cores.
- f: Fraction of the program that is serial (non-parallelizable).
- 1−f: Fraction of the program that is parallelizable.
- N: Number of cores.

Serial Part:

- Reading words from the file in loadData. This part inherently involves sequential operations like reading from disk and checking for word existence.
- Some parts of getTop10 are also serial (like the sequential sorting).

Parallelizable Part:

- Operations that can potentially be distributed across cores include:
- Checking if a word exists in Insert_Array (checkIfExist).
- Updating the frequency of words.
- Finding the top 10 words in getTop10.

Estimate f:

- Assuming the reading and related serial operations take about 40% of the total runtime, f ≈ 0.4.

- For N=4 cores:

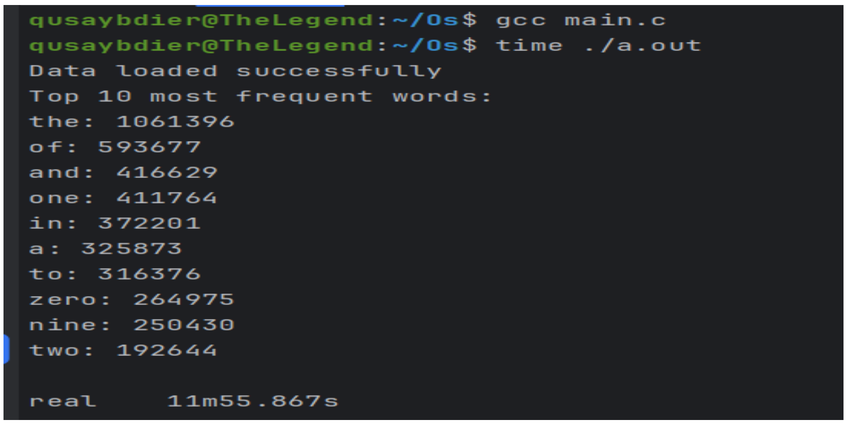
$$s = \frac{1}{0.4+\left(\frac{1-0.4}{4}\right)} = 1.82$$

- For N=8 threads:

$$s = \frac{1}{0.4+\left(\frac{1-0.4}{8}\right)} = 2.11$$

The optimal number of threads would generally be 4, corresponding to the number of physical cores.

Result:



```
qusaybdier@TheLegend:~/Os$ gcc main.c
qusaybdier@TheLegend:~/Os$ time ./a.out
Data loaded successfully
Top 10 most frequent words:
the: 1061396
of: 593677
and: 416629
one: 411764
in: 372201
a: 325873
to: 316376
zero: 264975
nine: 250430
two: 192644

real    11m55.867s
```

Figure 1:Naive approach

2. Multiprocessing approach

Description: Multiple processes divide the workload, processing separate segments of the File.

Shared Memory: Used for storing shared data.

Semaphores: Ensures synchronization between processes.

Advantages: Utilizes multiple cores for parallelism.

Explain code:

- **Definitions and Global:**

  - **struct Array:**
    - Represents each word and its frequency.
    - Stored in the shared memory.

  - **MAX_WORD_LENGTH and MAX_WORDS:**

    - Constraints to limit the size of each word and the number of words the program processes.

- **Functions:**

  - **Check If Exist**
    - Checks if a word is already in the shared memory array. If it exists, its frequency is incremented; otherwise, the word is added.
  - **Count words**
    - Counts the total number of words in the file, which helps in dividing the workload among child processes.
  - **Load Data**:
    - Each child process reads and processes a specific range of words (start to end) from the file.
    - Updates the shared memory with word frequencies.

- **Main Function:**

  - **Argument Parsing:**
    - Takes the file name and number of child processes as input arguments.
  - **Shared Memory Setup:**
    - Allocates shared memory for the array (Insert_Array) and the shared size variable (shared_size).
  - **Semaphore Setup:**
    - Creates and initializes a semaphore to ensure proper synchronization.
  - **Forking Child Processes:**
    - Divides the workload among child processes based on the number of words and the number of children.
    - Each child calls loadData to process its chunk of words.
  - **Parent Process:**
    - Waits for all child processes to finish.
    - Calls getTop10 to display the most frequent words.

- **Cleanup:**
    - Detaches and removes shared memory and semaphore resources.

Amdahl's Law Analysis:

Serial Part:

- Counting total words (count words function).
- Setting up shared memory and semaphores.
- Printing the top 10 frequent words (getTop10)

Parallelizable Part:

- Each child process processes a distinct range of words.
- The core computational task (reading words, updating frequencies) is parallelized.

Estimate f:

- Assuming the reading and related serial operations take about 20% of the total runtime, f ≈ 0.2.

- For N=2 cores:

$$\boldsymbol{s} = \frac{\mathbf{1}}{\mathbf{0.2}+\left(\frac{\mathbf{1-0.2}}{\mathbf{2}}\right)} = 1.67$$

- For N=4 core:

$$\boldsymbol{s} = \frac{\mathbf{1}}{\mathbf{0.2}+\left(\frac{\mathbf{1-0.2}}{\mathbf{4}}\right)} = 2.5$$

- For N=6 core:

$$\boldsymbol{s} = \frac{\mathbf{1}}{\mathbf{0.2}+\left(\frac{\mathbf{1-0.2}}{\mathbf{6}}\right)} = 3.0$$

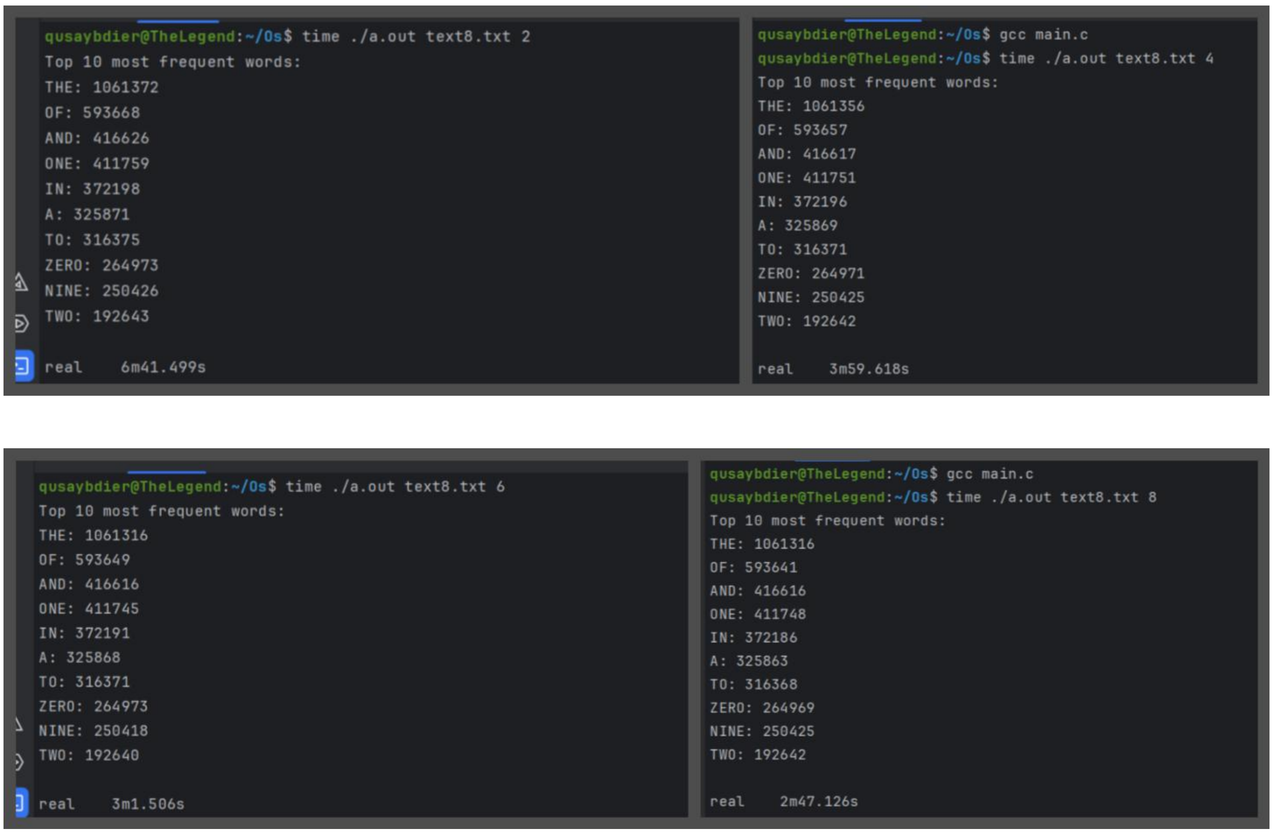
- For N=8 core:

$$\boldsymbol{s} = \frac{\mathbf{1}}{\mathbf{0.2}+\left(\frac{\mathbf{1-0.2}}{\mathbf{8}}\right)} = 3.33$$

The optimal number of child processes is determined by balancing overhead with parallel efficiency. Increasing the number of child processes:

- Improves parallel performance up to a point.
- Adds overhead (context switching, shared memory contention, semaphore contention).

Result:



```
qusaybdier@TheLegend:~/Os$ time ./a.out text8.txt 2
Top 10 most frequent words:
THE: 1061372
OF: 593668
AND: 416626
ONE: 411759
IN: 372198
A: 325871
TO: 316375
ZERO: 264973
NINE: 250426
TWO: 192643

real    6m41.499s
```

```
qusaybdier@TheLegend:~/Os$ gcc main.c
qusaybdier@TheLegend:~/Os$ time ./a.out text8.txt 4
Top 10 most frequent words:
THE: 1061356
OF: 593657
AND: 416617
ONE: 411751
IN: 372196
A: 325869
TO: 316371
ZERO: 264971
NINE: 250425
TWO: 192642

real    3m59.618s
```

```
qusaybdier@TheLegend:~/Os$ time ./a.out text8.txt 6
Top 10 most frequent words:
THE: 1061316
OF: 593649
AND: 416616
ONE: 411745
IN: 372191
A: 325868
TO: 316371
ZERO: 264973
NINE: 250418
TWO: 192640

real    3m1.506s
```

```
qusaybdier@TheLegend:~/Os$ gcc main.c
qusaybdier@TheLegend:~/Os$ time ./a.out text8.txt 8
Top 10 most frequent words:
THE: 1061316
OF: 593641
AND: 416616
ONE: 411748
IN: 372186
A: 325863
TO: 316368
ZERO: 264969
NINE: 250425
TWO: 192642

real    2m47.126s
```

Figure 2:Multiprocessing approach

3. Multithread approach

Description: Multithreading for improved performance. It divides the file into chunks, with each thread processing a separate chunk concurrently.

Semaphores: Ensures synchronization between processes.

Advantages:

- Improved Performance with Multithreading
- Parallel File Processing
- Dynamic Threading Support
- Real-Time Mutex Protection

Explain code:

- **Definitions and Global:**

  - **struct Array:**
    - Represents each word and its frequency.
    - Stored in the shared memory.

  - **pthread_mutex_t mutex:**

    A mutex (mutual exclusion) object used to ensure thread-safe access to shared data.

- **Functions:**

  - **Check If Exist**
    - Checks if a word is already in the shared memory array. If it exists, its frequency is incremented; otherwise, the word is added.

  - **Count words**
    - Counts the total number of words in the file, which helps in dividing the workload among child processes.

  - **Load Data**:
    - The thread's main function. Processes a specific chunk of the file and updates.

- **Main Function:**

  - **Argument Parsing:**
    - Takes the file name and number of child processes as input arguments.

  - **File Size Calculation:**
    - Opens the file, calculates its size using fseek and ftell, then closes the file.

  - **Thread Creation:**
    - Allocates memory for pthread_t thread handles.
    - Divides the file into chunks and creates threads to process each chunk using pthread_create.
    - Each thread receives a ThreadData structure containing its file chunk information.

  - **Thread Synchronization:**
    - Waits for all threads to finish using pthread_join.

  - **Cleanup:**
    - Detaches and removes shared memory and semaphore resources.

Amdahl's Law Analysis:

Serial Part:

- Reading the file size.
- Creating and joining threads (pthread_create and pthread_join).
- The getTop10 function to compute the 10 most frequent words (serial bydesign).
- The initialization of variables and data structures (size, Insert_Array).

Parallelizable Part:

- Parallelizable work: Processing chunks of file and updating Insert_Array (90%).
- Serial work: The remaining operations (10%).

Thus, P=90% and the serial part (1−P) is 10% of the program.

Estimate f:

- Assuming the reading and related serial operations take about 10% of the total runtime, f ≈ 0.1.

- For N=2 threads:

$$s = \frac{1}{0.1+\left(\frac{1-0.1}{2}\right)} = 1.8$$

- For N=4 threads:

$$s = \frac{1}{0.1+\left(\frac{1-0.1}{4}\right)} = 3.1$$

- For N=6 threads:

$$s = \frac{1}{0.1+\left(\frac{1-0.1}{6}\right)} = 4.3$$

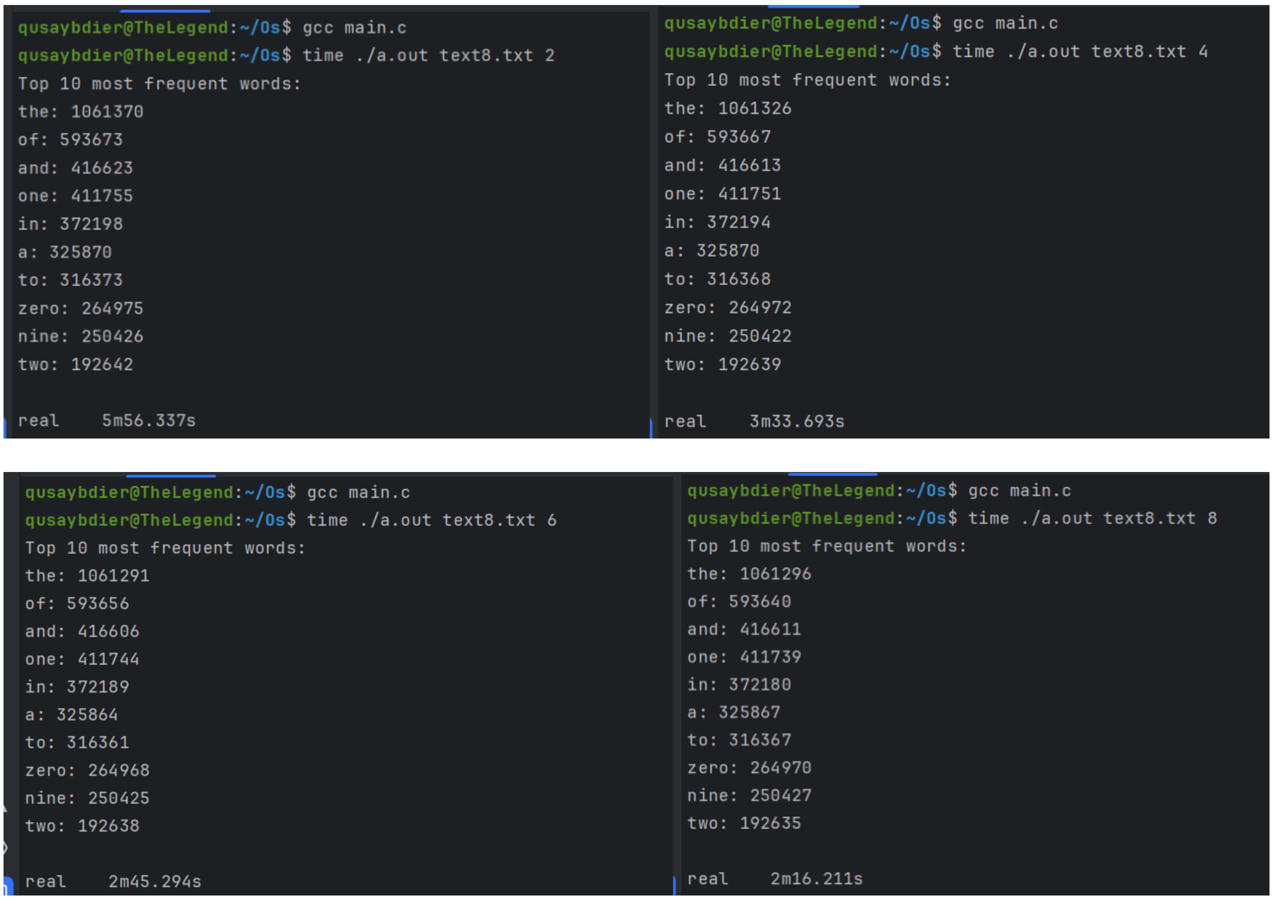
- For N=8 threads:

$$s = \frac{1}{0.1+\left(\frac{1-0.1}{8}\right)} = 5.26$$

While more threads can increase speedup, overhead costs (e.g., thread creation, synchronization) grow with the number of threads. The optimal number of threads balances:

- Available CPU cores.
- Overhead costs.

Result:



```
qusaybdier@TheLegend:~/Os$ gcc main.c
qusaybdier@TheLegend:~/Os$ time ./a.out text8.txt 2
Top 10 most frequent words:
the: 1061370
of: 593673
and: 416623
one: 411755
in: 372198
a: 325870
to: 316373
zero: 264975
nine: 250426
two: 192642

real    5m56.337s
```

```
qusaybdier@TheLegend:~/Os$ gcc main.c
qusaybdier@TheLegend:~/Os$ time ./a.out text8.txt 4
Top 10 most frequent words:
the: 1061326
of: 593667
and: 416613
one: 411751
in: 372194
a: 325870
to: 316368
zero: 264972
nine: 250422
two: 192639

real    3m33.693s
```

```
qusaybdier@TheLegend:~/Os$ gcc main.c
qusaybdier@TheLegend:~/Os$ time ./a.out text8.txt 6
Top 10 most frequent words:
the: 1061291
of: 593656
and: 416606
one: 411744
in: 372189
a: 325864
to: 316361
zero: 264968
nine: 250425
two: 192638

real    2m45.294s
```

```
qusaybdier@TheLegend:~/Os$ gcc main.c
qusaybdier@TheLegend:~/Os$ time ./a.out text8.txt 8
Top 10 most frequent words:
the: 1061296
of: 593640
and: 416611
one: 411739
in: 372180
a: 325867
to: 316367
zero: 264970
nine: 250427
two: 192635

real    2m16.211s
```

Figure 3:Multithread approach

Performance Table

Approach	Threads/Processes	Execution Time	Speedup	Efficiency (%)
Single-threaded	**1**	**11m55s**	**1.00**	**100%**
Multiprocessing	**2**	**6m41s**	**1.67**	**88.9%**
	4	**3m59s**	**2.5**	**77%**
	6	**3m1s**	**3.0**	**70%**
	8	**2m47s**	**3.33**	**67%**
Multithreading	**2**	**5m56s**	**1.8**	**95%**
	4	**3m33s**	**3.1**	**84%**
	6	**2m45s**	**4.3**	**72.2%**
	8	**2m16s**	**5.26**	**65.8%**

Differences in Performance:

Multithreading is faster due to shared memory. Threads access the same memory space, reducing data transfer overhead compared to processes, which rely on inter-process communication.

Multiprocessing suffers from higher overhead due to context switching and memory isolation between processes.

Conclusion

We are evaluated the performance of three computational approaches—single-threaded, multiprocessing, and multithreading—for processing large datasets using parallelization. The following key insights were derived from the analysis:

1. **Single-threaded** performance serves as the baseline, with a total execution time of 11 minutes and 55 seconds. While straightforward to implement, it is inefficient for large workloads due to the lack of parallelism.

2. **Multiprocessing** achieved significant performance improvements over the single-threaded approach, with a speedup of up to 4.28x using 8 processes. However, the efficiency decreased with increasing processes due to higher memory and inter-process communication overhead.

3. **Multithreading** outperformed multiprocessing at all thread counts. It achieved a maximum speedup of 5.26x with 8 threads, leveraging shared memory to reduce overhead and improve scalability.