Faculty of Engineering & Technology Electrical & Computer Engineering Department

Operating Systems Concepts ENCS3390

Project 1 Report

Naive, Multiprocessing and Multithreading Approaches

Prepared by:

Qasim Batrawi 1220204

Instructor: Dr. Yazan Abu Farha

Section: 1

Date: 27-11-2024

Abstract

The project explores the differences between three different approaches: Naive, Multiprocessing and Multithreading. It examines how each approach handles to the same task, explains how they work, and compares their execution time. The goal is to find out which approach is the most efficient.

Table of Contents

List of Figures

List of Tables

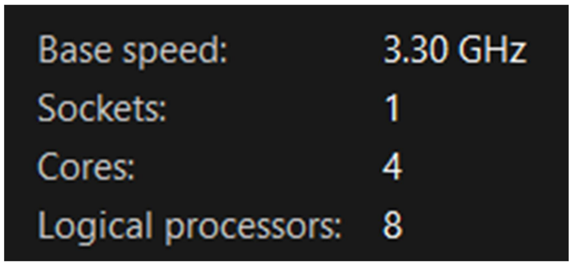
1 Overview

In this project, we have built a program to count the frequency of each word in a file. Naive, Multiprocessing and Multithreading were used to compare the performance and time efficiency of each method.

The Naive approach processes the file sequentially, while Multiprocessing and Multithreading approaches will split the tasks into multiple processes, which will allow parallel work and better CPU utilization.

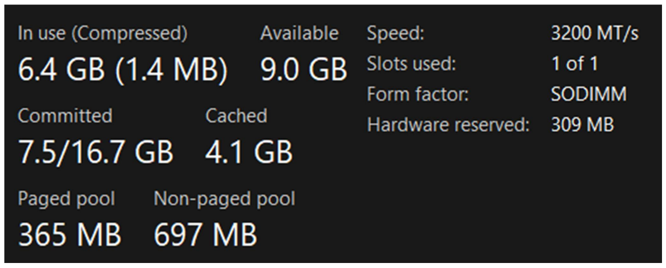
2 System Configuration and Development Environment

My Device has four core processor with eight logical processors, a speed of 3.3 GHz, and a 16 GB of RAM.



Base speed:	3.30 GHz
Sockets:	1
Cores:	4
Logical processors:	8

Figure 1 My CPU.



In use (Compressed)	6.4 GB (1.4 MB)
Available	9.0 GB
Committed	7.5/16.7 GB
Cached	4.1 GB
Paged pool	365 MB
Non-paged pool	697 MB
Speed:	3200 MT/s
Slots used:	1 of 1
Form factor:	SODIMM
Hardware reserved:	309 MB

Figure 2 My Memory

I have used Linux operating system, installed on a flash device, with a full access to my computer's cores and memory without any limitations. For programming, I worked with C language using Code::Blocks IDE.

3 Functionality of My Code

Here are the steps that I have followed in the three approaches

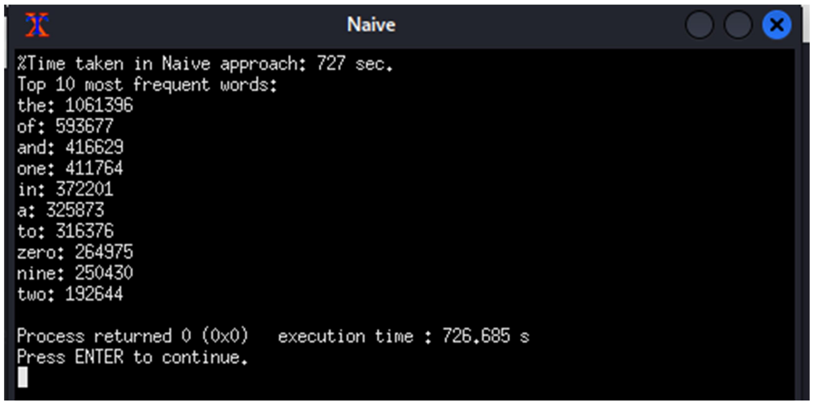
1. Extract all the words from the file and store them in the **allWords** array.
2. Compute the frequency of each word stored in **allWords** array and store the results in the **uniqueWords** array, which contains each unique word along with its frequency.
3. Find the top 10 most frequent words stored in **uniqueWords** array, by just finding the top 10 without sorting the entire array.

In Naive approach, the steps will run sequentially. In Multiprocessing and Multithreading approaches

- Step 1 runs sequentially.
- Step 2 runs in a parallel using child processes or threads.
- Step 3 runs sequentially.

4 Naive Approach

Naive approach solves the task using simple straight-forward code, as it runs with only one process at a time. This is the most basic method that a programmer would use, as it is the simplest way to implement the solution. We have got the following result with Naive approach.



```
%Time taken in Naive approach: 727 sec.
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

Process returned 0 (0x0)   execution time : 726.685 s
Press ENTER to continue.
```

Figure 3 Naive approach results.

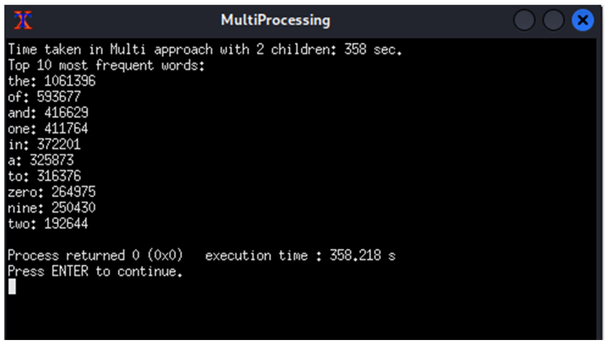
Using this approach, it took 727 seconds or approximately 12 minutes, to calculate and print the top ten most frequent words. Time is quite long, we will try to reduce it by using Multiprocessing and Multithreading, so more than one process or thread will run in parallel.

5 Multiprocessing Approach

In Multiprocessing, we can make more than one process to run in parallel, which can help to speed up the time needed. This is done using fork() system call, which creates a child process from the currently running process. The child process can be indicated by the returned value from fork(), which will be zero.

A shared memory has been used, where multiple processes can write on the same memory space. To avoid race conditions, we have used synchronization using the built-in __sync_lock_test_and_set() function to acquire the lock, and the built-in __sync_lock_release() function to release it. So if a process attempts to write on the shared memory, it first takes the lock, causing other children to wait until the lock is released. The parent process will wait for its children to terminate using wait() system call. Other APIs has been used, such as shmget(), which is used to create the shared memory.

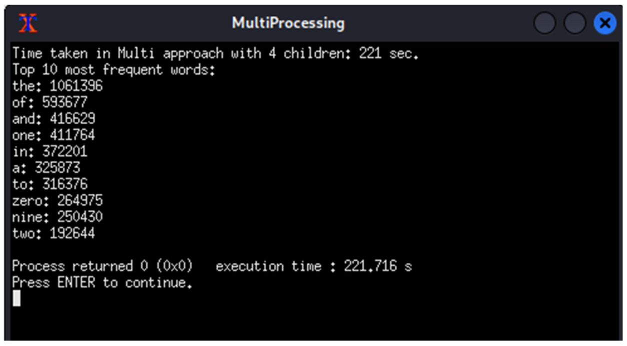
We ran the program four times with 2, 4, 6 and 8 child processes to notice the difference. The following results were obtained.



```
Time taken in Multi approach with 2 children: 358 sec.
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

Process returned 0 (0x0)   execution time : 358.218 s
Press ENTER to continue.
```

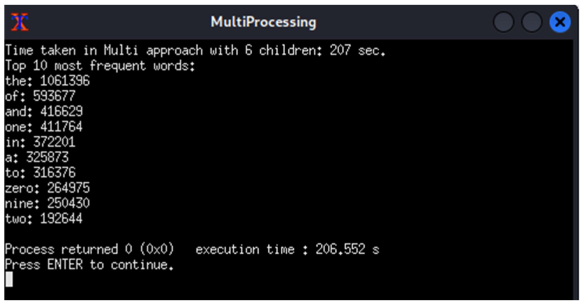
Figure 5 Results with 2 child processes.



```
Time taken in Multi approach with 4 children: 221 sec.
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

Process returned 0 (0x0)   execution time : 221.716 s
Press ENTER to continue.
```

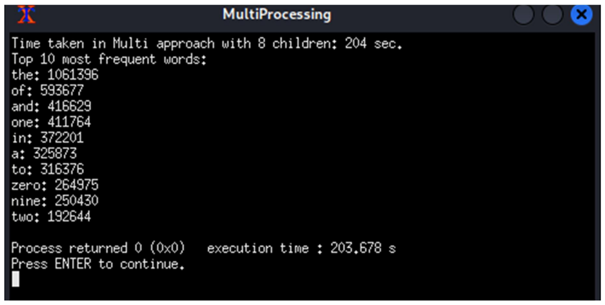
Figure 4 Results with 4 child processes.



```
MultiProcessing
Time taken in Multi approach with 6 children: 207 sec.
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

Process returned 0 (0x0)   execution time : 206.552 s
Press ENTER to continue.
```

Figure 7 Results with 6 child processes.



```
MultiProcessing
Time taken in Multi approach with 8 children: 204 sec.
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

Process returned 0 (0x0)   execution time : 203.678 s
Press ENTER to continue.
```

Figure 6 Results with 8 child processes.

As shown in figures, there is a noticeable improvement on the time comparing to Naive approach. With 2 child processes, it took 358 seconds (6 minutes), with 4 child processes, it took 221 seconds (3.6 minutes), with 6 child processes, it took 207 seconds (3.45 minutes), and with 8 child processes, it took 204 seconds (3.4 minutes).

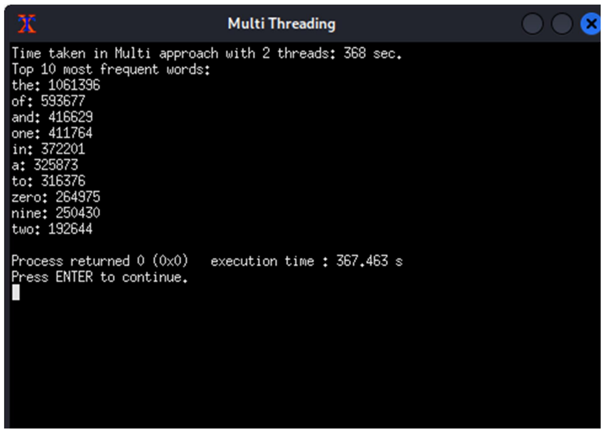
We can notice that the difference decreases as the number of processes increases, and this due to the overhead (e.g., context switch and synchronization).

6 Multithreading Approach

In Multithreading, multiple threads will run in parallel, with a shared data, file, and code between them. This done by using pthread_create() API to create and run the thread to its function. The process will wait for the termination of its threads using pthread_join().

A mutex lock has been used to avoid race conditions, since the same data is shared between the threads. The thread can acquire the lock using the built-in pthread_mutex_lock() function, and releases the lock using the built-in pthread_mutex_unlock() function.

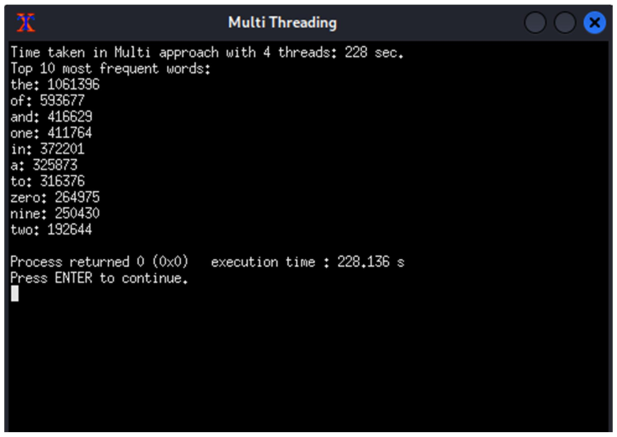
We ran the program four times with 2, 4, 6 and 8 thread processes to notice the difference. The following results were obtained.



```
Time taken in Multi approach with 2 threads: 368 sec.
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

Process returned 0 (0x0)   execution time : 367.463 s
Press ENTER to continue.
```

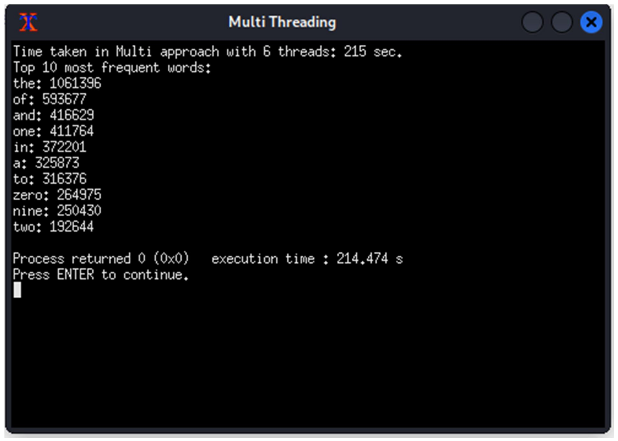
Figure 8 Results with 2 thread processes.



```
Time taken in Multi approach with 4 threads: 228 sec.
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

Process returned 0 (0x0)   execution time : 228.136 s
Press ENTER to continue.
```

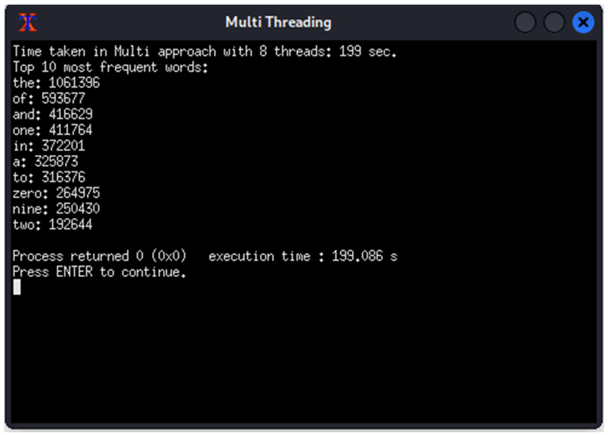
Figure 9 Results with 4 thread processes.



```
Time taken in Multi approach with 6 threads: 215 sec.
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

Process returned 0 (0x0)   execution time : 214.474 s
Press ENTER to continue.
```

Figure 10 Results with 6 thread processes.



```
Time taken in Multi approach with 8 threads: 199 sec.
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

Process returned 0 (0x0)   execution time : 199.086 s
Press ENTER to continue.
```

Figure 11 Results with 8 thread processes.

As shown in figures, there is a noticeable improvement on the time comparing to Naive approach. With 2 threads, it took 368 seconds (6 minutes), with 4 threads, it took 228 seconds (3.8 minutes), with 6 threads, it took 215 seconds (3.5 minutes), and with 8 threads, it took 199 seconds (3.3 minutes).

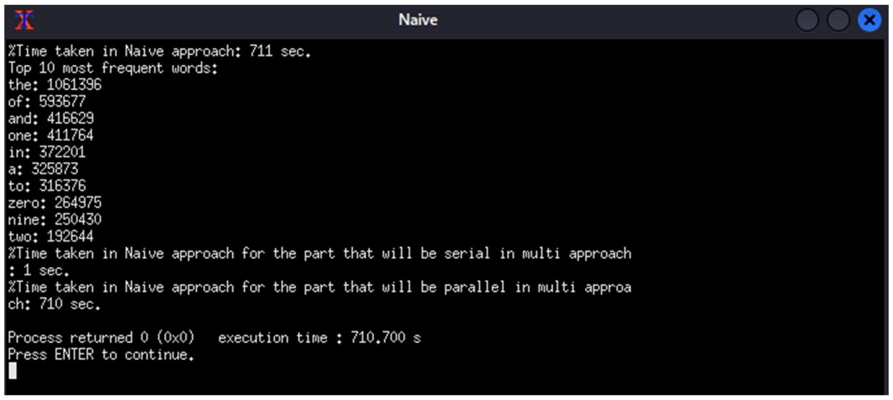
7 Analysis to Amdahl's Law

Amdahl's Law highlights that the serial portion of an application has disproportionate effect on performance gained by adding additional cores. It identifies performance gains from adding additional cores to an application that has both serial and parallel components. The following equation describes the speed up.

$$Speed\ up = \frac{1}{S + \frac{1 - S}{N}}$$

Where S is the serial portion and N is the number of logical cores.

We ran the Naive approach again to calculate the serial portion. The following result shows the time taken for the serial parts, corresponding to steps 1 and 3 here.



```
%Time taken in Naive approach: 711 sec.
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
%Time taken in Naive approach for the part that will be serial in multi approach
: 1 sec.
%Time taken in Naive approach for the part that will be parallel in multi approa
ch: 710 sec.

Process returned 0 (0x0)   execution time : 710.700 s
Press ENTER to continue.
```

Figure 12 Serial portion in Naive approach.

As shown in the figure, the serial parts take 1 second out of total 711 seconds. The speed up can be calculated as follows.

$$S = \frac{1}{711} = 1.406 \times 10^{-3} = 0.1406\%$$

$$1 - S = 0.99859 = 99.859\%$$

$$N = 8$$

$$speed\ up = \frac{1}{1.406 \times 10^{-3} + \frac{0.99859}{8}} = 7.922$$

The optimal number of processes or threads depends on the number of CPU cores. With my 8 logical cores, the optimal number of processes or threads is 8, which ensures that there won't be too much time spent on context switching.

8 Comparison Table

Approach		Execution Time
Naive		727 seconds
Multiprocessing	2 children	358 seconds
	4 children	221 seconds
	6 children	207 seconds
	8 children	204 seconds
Multithreading	2 threads	368 seconds
	4 threads	228 seconds
	6 threads	215 seconds
	8 threads	199 seconds

Table 1 Comparsion of the three approaches.

It's clear from the table that Multiprocessing and Multithreading are faster than the Naive approach, since more than one process or thread work at one time. Comparing between Multiprocessing and Multithreading, it seems that the two approaches are close to each other in their execution time. However, Multithreading may be a better choice than Multiprocessing, since it does not require a shared memory, overhead is reduced.

9 Conclusion

To conclude, the project compared between the Naive, Multiprocessing, and Multithreading. Results showed that the Multi approaches are much better in execution time than Naive approach. While Multiprocessing and Multithreading has similar execution time with small differences, Multithreading proved to be a better choice due to the overhead in Multiprocessing.

We have faced a few problems in the communication between the processes in Multiprocessing, we attempted to use pipes. However, the buffer in pipes has a limited size, so due to the large data, pipes can’t be used. As a result, we have used a shared memory with synchronization.

Overall, the project gave us a good knowledge on how to operate with different programming approaches, giving us a better understand to write an efficient code.