Faculty of Engineering and Technology
Electrical and Computer Engineering Department

OPERATING SYSTEMS
ENCS3390

Project1

Prepared by:
Student Name: leen alqazaqi **Student NO. 1220380**

Instructor: **Yazan abu Faraha**

Section: **1**

Abstract

In this project we worked on finding the top 10 most frequent words in a large data set using three approaches, naïve, multiprocessing, and multithreading. We have found out that the multithreading code is the best one (the fastest) then the multiprocessing and lastly the naïve. So, by this results we learned the importance of using multiprocessing or multithreading.

Table of Contents

Environment

Description for the environment:

Cores: 4 physical cores, 8 virtual

Memory: 16 GB ram

Speed (clock speed): 2.40GHz

OS: My laptop runs windows, so I installed the Windows Subsystem for Linux (WSL) to use the Linux terminal and the system calls from the Linux.

Programming language: C language was used.

IDE Tool: I used CLion (CLion- JetBrains) program with the WSL environment to run the code.

No virtual machine was used; only a Linux subsystem on Windows was utilized.

Procedure:

1.1 How multiprocessing and multitasking were achieved

First, I would like to discuss the general algorithm I used. The project has three main steps, reading from the file, counting word frequencies and sorting.

I chose to read the data into a linked list since the size is unknown at the beginning. Once all the data is read, the linked list elements were moved into an array of structs.

After reading we started the counting step. This is the most important part of the code since it is the part, I decided to divide it between child processes and threads (in the multiprocessing and multithreading parts). Unique words were moved into a smaller array and I checked through both of the array if the word exists in the unique array the count for it increases, if not, the word is added to the array. This approach has a time complexity of O(m*n), where m is the number of unique words and n is the total number of words. It is better than n^2 but at the same time it shows the importance of using multiprocessing and multithreading approaches.

Finally, after counting, I used merge sort to sort the smaller array by frequency to find the top 10 most frequent words.

1.1.1 Multiprocessing implementation:

The multiprocessing is achieved using the fork system call (which is one of the Linux OS system calls) in combination with IPC using pipes (each child has its own pipe).

The multiprocessing is for the calculating frequency part where each child takes a chunk of data and calculate the frequency for it, then the results of the child processes is merged. This method is also known as data parallelism where the data is divided between different processes each work on its part on different core.

Note that the used pipes are ordinary pipes that are efficient in the condition of communication between parent and child as in our case.

API and functions used:

`pipe`: a function that Creates a unidirectional data channel used for IPC between the parent and child processes.

`fork`: a system call that creates child processes to work in parallel

`write` and `read`: a function used to transfer data between parent and child processes through the pipes.

`wait`: a system call used by the parent process to ensure that all child processes finished execution before the parent continues.

Also, malloc and free functions were used to dynamically allocate memory.

Functions used in my code to find the result of the multiprocessing approach:

```
struct sortedNodes *MultiProcessing(struct sortedNodes
*TheWords, int count, int numofchildren);
```

it takes the array of words as a parameter with the number of words in the array and the number of children we want to create

this function divides the data into chunks create children and calls the frequency calculation function for each chunk. After getting the results it calls the following function that works on merging the results of all child processes.

```
void MergeChildrenResults(struct sortedNodes *final, struct
sortedNodes *childResults, int *finalCount, int childCount);
```

this code takes a pointer to all merged results from all child processes as a parameter. In addition to a pointer for an array having all the child process results and a pointer to an integer that tracks the total number of unique words in final. Finally, it also takes the number of elements in the child results array.

//the frequency calculation function

`void multiprocessingFrequency(struct sortedNodes *chunk, int count, int PipeWrite);`

this function takes a pointer to the chunk the child is working on, the number of the words in the chunk, and a file descriptor to write on the write end of the pipe. It counts the frequency of the words in the chunk.

How multiprocessing is done:

After reading the file sequentially as described previously, the data is divided into equal sized chunks for each child (each child takes one of these equal sized chunks).

Each child will find the frequency for the words in its chunk and sends the results back for its parent through the pipe.

The parent reads the results the child processes wrote on the pipe to produce a list with all the words and their frequencies

The final result is sorted and the top 10 most frequent words are printed (this part is done sequentially).

Note that the multiprocessing uses the same algorithm as in the naïve code. The difference is in dividing the data into chunks and each child take one. This causes the need of communication between processes that’s why the child writes on its side of the pipe and the parent reads from its side

The code segment where the parent reads from the pipe :

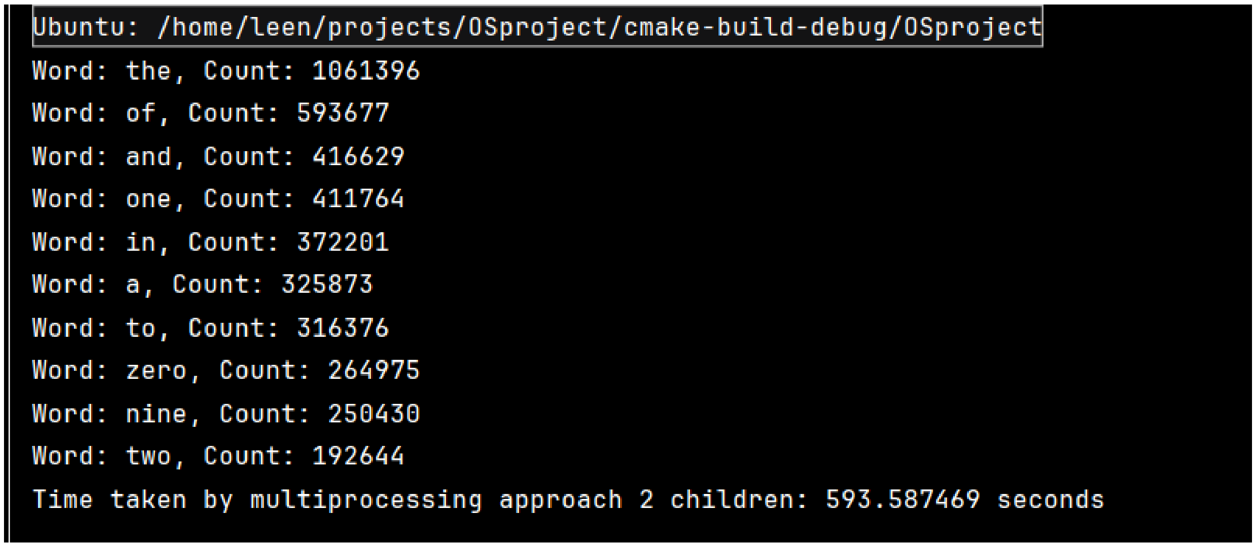
```
for (int j = 0; j < local; j++) {
    int wordLength;
    read(OrdPipes[i][0], &wordLength, sizeof(wordLength));
    childResults[j].word = malloc(wordLength);
    read(OrdPipes[i][0], childResults[j].word, wordLength);
    read(OrdPipes[i][0], &childResults[j].count,
sizeof(childResults[j].count));}
close(OrdPipes[i][0]);
```

The code segment where the child writes its result into the pipe :

```c
write(PipeWrite, &local, sizeof(local));
for (int i = 0; i < local; i++) {
    int wordLength = strlen(uniqueWords[i].word) + 1;
    write(PipeWrite, &wordLength, sizeof(wordLength));
    write(PipeWrite, uniqueWords[i].word, wordLength);
    write(PipeWrite, &uniqueWords[i].count, sizeof(uniqueWords[i].count));
    free(uniqueWords[i].word);
}
```

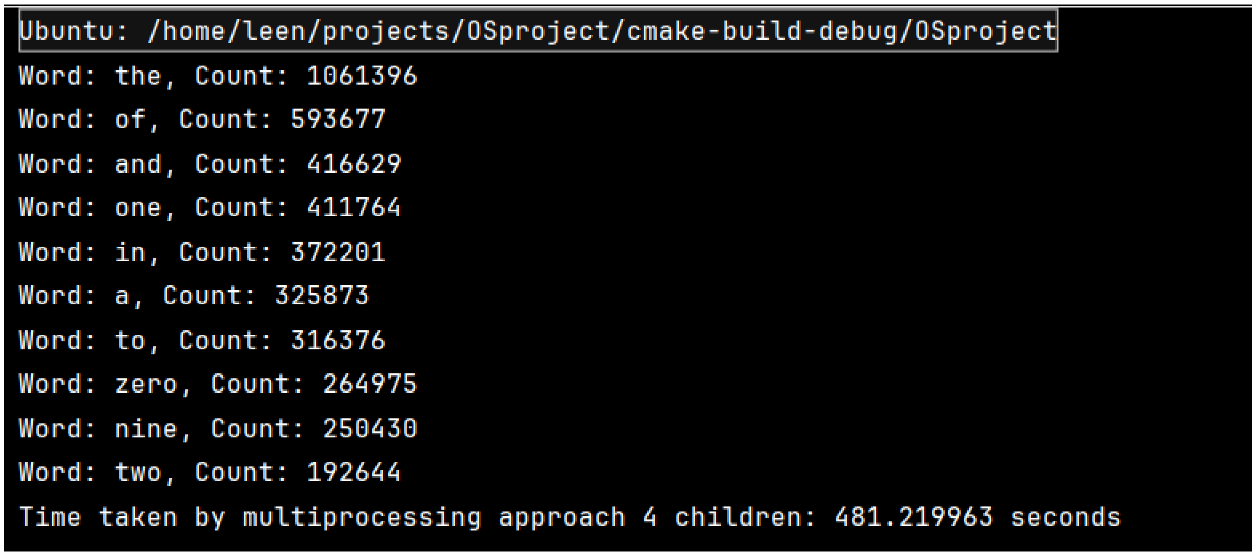
Results for multiprocessing : below is the results for running multiprocessing code for 2,4,6 and 8 child processes (the top 10 words in addition to the execution time)

Two child processes:



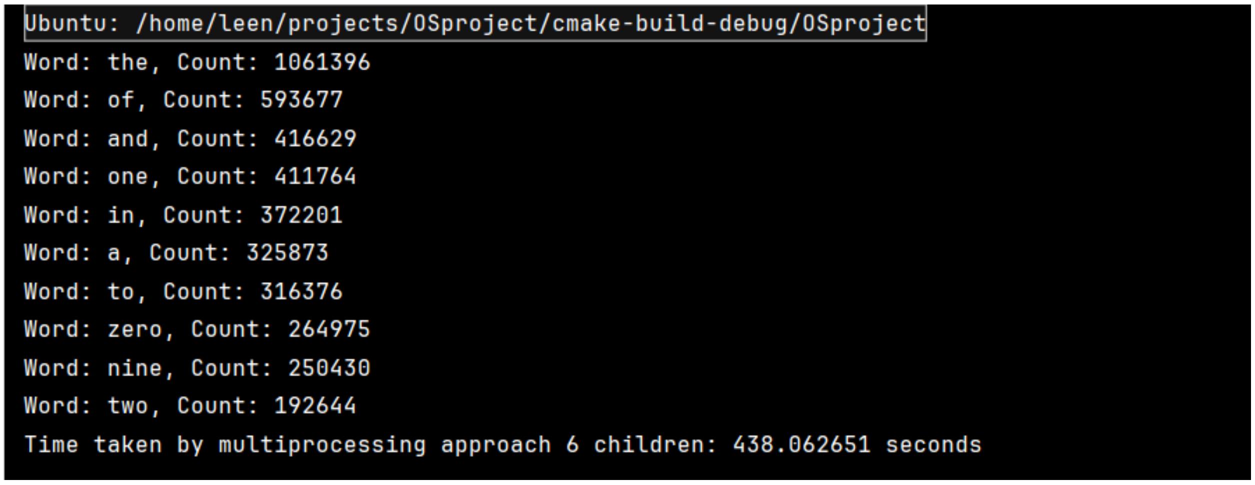
```
Ubuntu: /home/leen/projects/OSproject/cmake-build-debug/OSproject
Word: the, Count: 1061396
Word: of, Count: 593677
Word: and, Count: 416629
Word: one, Count: 411764
Word: in, Count: 372201
Word: a, Count: 325873
Word: to, Count: 316376
Word: zero, Count: 264975
Word: nine, Count: 250430
Word: two, Count: 192644
Time taken by multiprocessing approach 2 children: 593.587469 seconds
```

Four child processes:



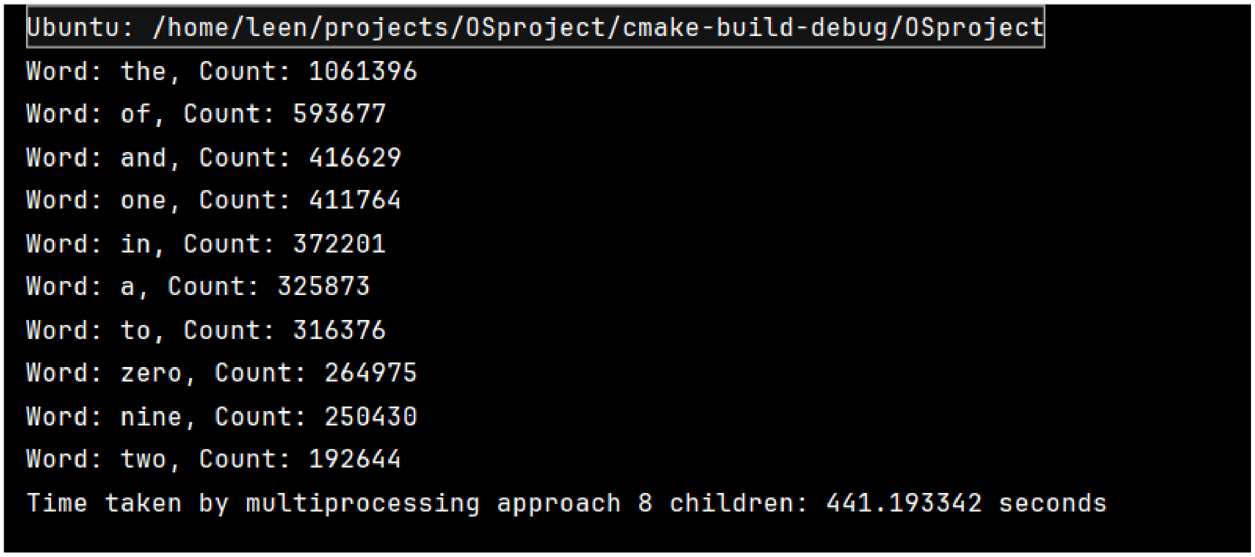
```
Ubuntu: /home/leen/projects/OSproject/cmake-build-debug/OSproject
Word: the, Count: 1061396
Word: of, Count: 593677
Word: and, Count: 416629
Word: one, Count: 411764
Word: in, Count: 372201
Word: a, Count: 325873
Word: to, Count: 316376
Word: zero, Count: 264975
Word: nine, Count: 250430
Word: two, Count: 192644
Time taken by multiprocessing approach 4 children: 481.219963 seconds
```

6 child processes:



```
Ubuntu: /home/leen/projects/OSproject/cmake-build-debug/OSproject
Word: the, Count: 1061396
Word: of, Count: 593677
Word: and, Count: 416629
Word: one, Count: 411764
Word: in, Count: 372201
Word: a, Count: 325873
Word: to, Count: 316376
Word: zero, Count: 264975
Word: nine, Count: 250430
Word: two, Count: 192644
Time taken by multiprocessing approach 6 children: 438.062651 seconds
```

8 child processes:



```
Ubuntu: /home/leen/projects/OSproject/cmake-build-debug/OSproject
Word: the, Count: 1061396
Word: of, Count: 593677
Word: and, Count: 416629
Word: one, Count: 411764
Word: in, Count: 372201
Word: a, Count: 325873
Word: to, Count: 316376
Word: zero, Count: 264975
Word: nine, Count: 250430
Word: two, Count: 192644
Time taken by multiprocessing approach 8 children: 441.193342 seconds
```

1.1.2 Multithreading implementation**:**

The multithreading is implemented using the POSIX threads (pthreads) library to divide the work among the threads. A mutex is used as a lock to prevent the race condition since there is some shared resources.

API and functions used:

`pthread_create`: a function that creates threads to execute the frequency calculation.

`pthread_join`: a function to wait for threads to complete execution.

`pthread_mutex_lock` and **`pthread_mutex_unlock`:** functions that are used to prevent race conditions when accessing shared resources. They are used to control the mutex (to lock it or unlock it).

As in the previous part the malloc and free are used to dynamically allocate memory

The functions in my code to get the multithreading result

`struct sortedNodes *MultiThreading(struct sortedNodes *TheWords, int count, int numofthreads);`

takes the array with the words as a parameter with its size and the number of the threads

inside the previous function this function is called

`void *threadFrequencyCalculation(void *args);`

this function takes a pointer to the ThreadArgs struct as a parameter; this struct contains a pointer to the data chunk this specific thread is working on. It also has the size of the chunk stored in it, a pointer to the shared result, a pointer the total unique words count and a pointer to the mutex that controls merging and prevent race condition.

How multithreading is done:

After reading the file sequentially as described previously, the data is divided into equal sized chunks each is processed by a separate thread.

Each thread calculates the frequency of the words in its chunk and stores the results locally.

The threads use a mutex to merge the results and at the same time o prevent any possibly race conditions. Which results into a shared result.

After all threads complete the execution, the final shared result is sorted and the top 10 most frequent words are printed (this part is sequential and no threads are used).

Note that the algorithm used in the multithreading is similar to the one in the naïve the difference is that some functions in the POSIX library are used to create threads each one takes a chunk of the whole data.

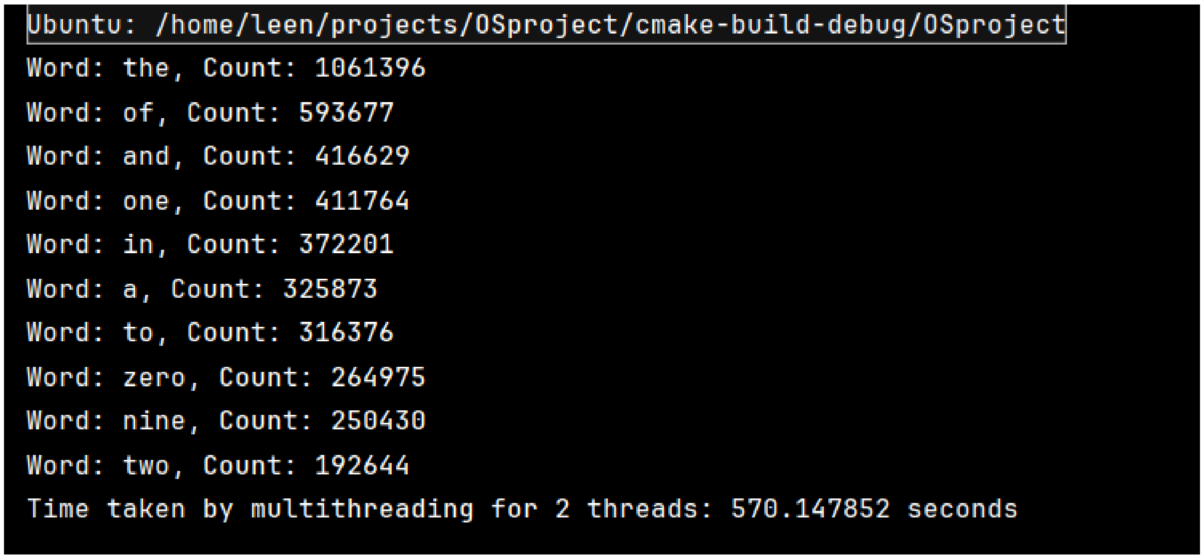
Here is the code that creates the mutex and divides the data into chunks:

```c
pthread_mutex_t mutex;
pthread_mutex_init(&mutex, NULL);
//divide the data into chunks each worked on by a thread
for (int i = 0; i < num_threads; i++) {
    args[i].chunk = TheWords + i * chunk_size;
    if (i == num_threads - 1) {
        //  printf("test");
        args[i].chunkSize = count - i * chunk_size;
    } else {
        args[i].chunkSize = chunk_size;
    }
    args[i].result = FrequncyCalculated;
    args[i].finalCount = &finalCount;
    args[i].mutex = &mutex;

    pthread_create(&threads[i], NULL, threadFrequencyCalculation, &args[i]);
}
```

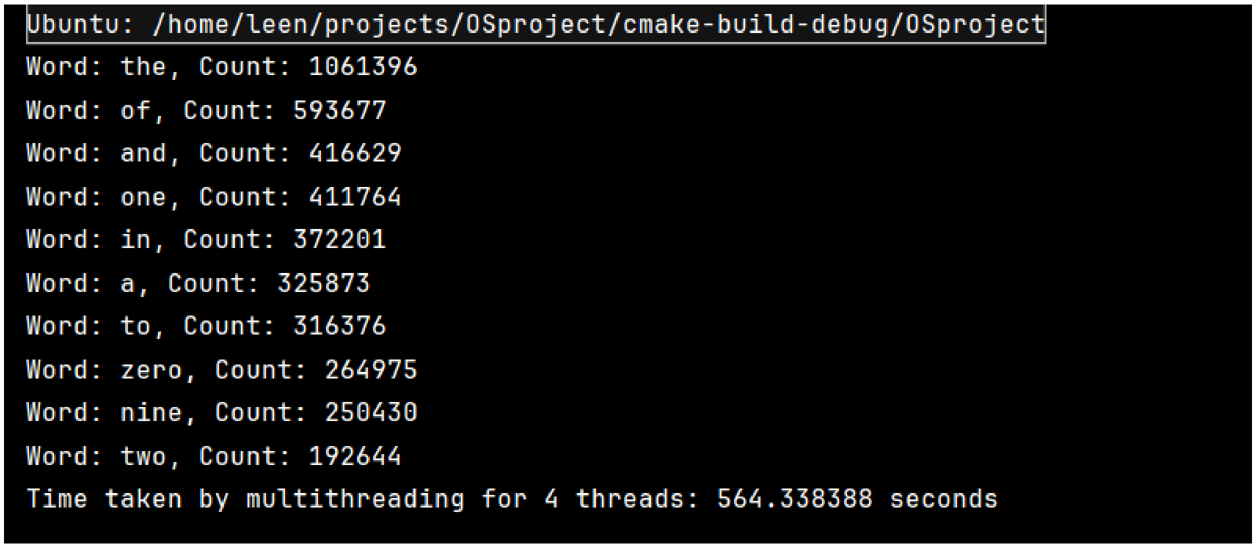
Results for multithreading below is the results for running multithreading code for 2,4,6 and 8 threads (the top 10 words in addition to the execution time)

2 threads:



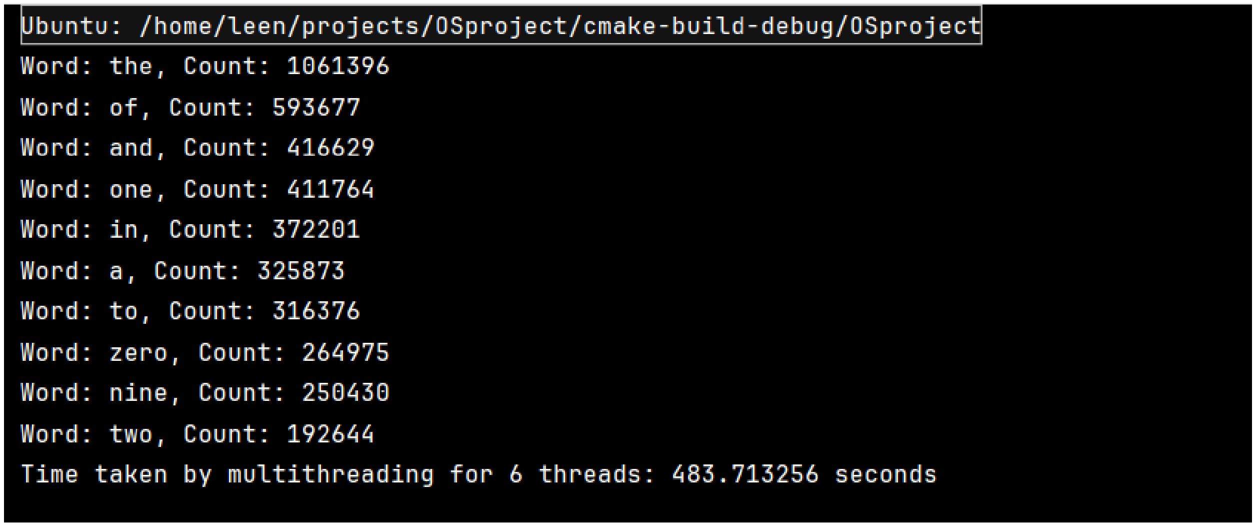
```
Ubuntu: /home/leen/projects/OSproject/cmake-build-debug/OSproject
Word: the, Count: 1061396
Word: of, Count: 593677
Word: and, Count: 416629
Word: one, Count: 411764
Word: in, Count: 372201
Word: a, Count: 325873
Word: to, Count: 316376
Word: zero, Count: 264975
Word: nine, Count: 250430
Word: two, Count: 192644
Time taken by multithreading for 2 threads: 570.147852 seconds
```

4 threads:

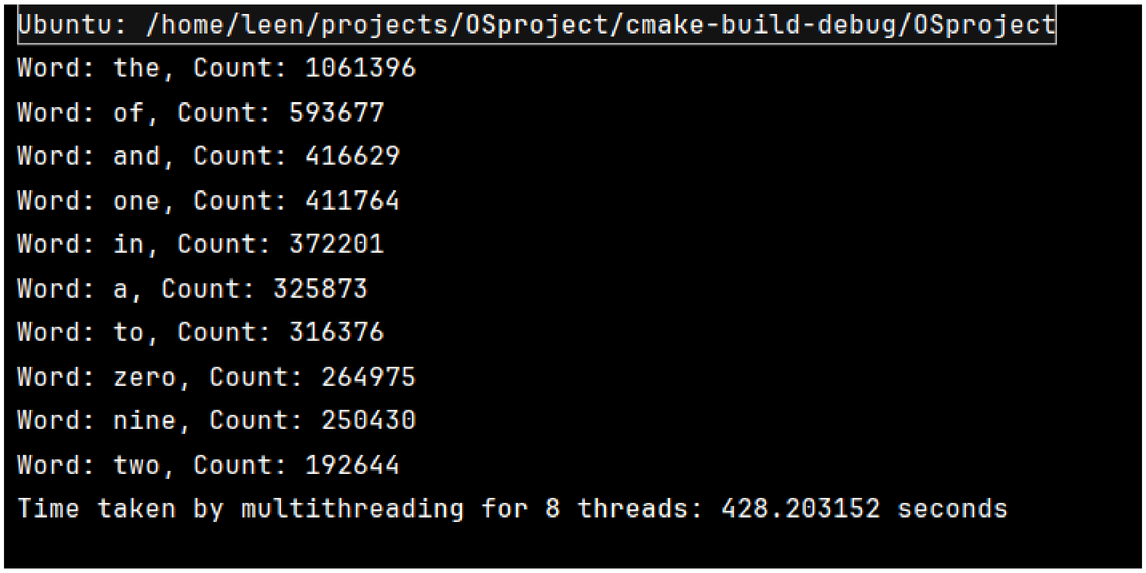


```
Ubuntu: /home/leen/projects/OSproject/cmake-build-debug/OSproject
Word: the, Count: 1061396
Word: of, Count: 593677
Word: and, Count: 416629
Word: one, Count: 411764
Word: in, Count: 372201
Word: a, Count: 325873
Word: to, Count: 316376
Word: zero, Count: 264975
Word: nine, Count: 250430
Word: two, Count: 192644
Time taken by multithreading for 4 threads: 564.338388 seconds
```

6 threads:


```
Ubuntu: /home/leen/projects/OSproject/cmake-build-debug/OSproject
Word: the, Count: 1061396
Word: of, Count: 593677
Word: and, Count: 416629
Word: one, Count: 411764
Word: in, Count: 372201
Word: a, Count: 325873
Word: to, Count: 316376
Word: zero, Count: 264975
Word: nine, Count: 250430
Word: two, Count: 192644
Time taken by multithreading for 6 threads: 483.713256 seconds
```

8 threads:


```
Ubuntu: /home/leen/projects/OSproject/cmake-build-debug/OSproject
Word: the, Count: 1061396
Word: of, Count: 593677
Word: and, Count: 416629
Word: one, Count: 411764
Word: in, Count: 372201
Word: a, Count: 325873
Word: to, Count: 316376
Word: zero, Count: 264975
Word: nine, Count: 250430
Word: two, Count: 192644
Time taken by multithreading for 8 threads: 428.203152 seconds
```

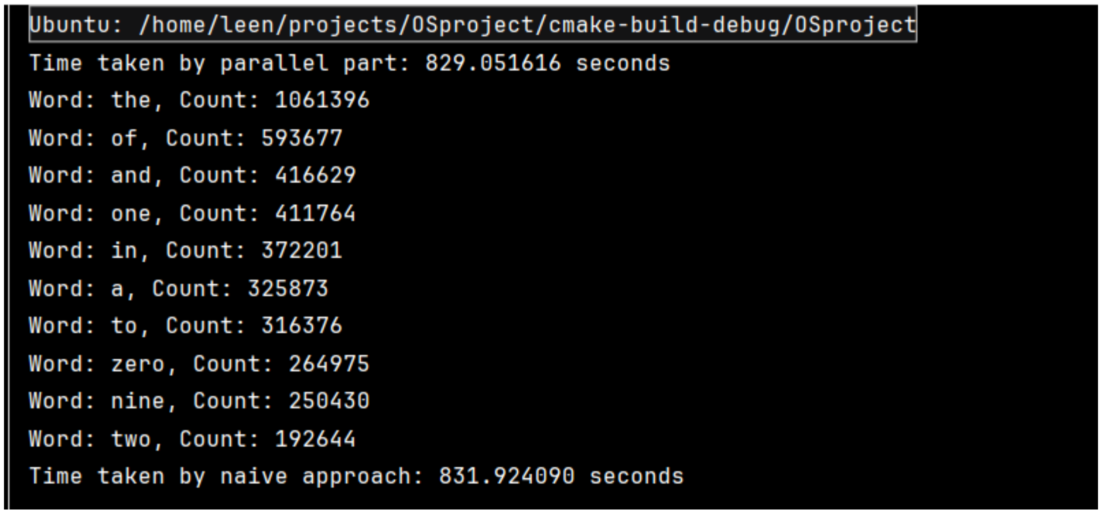
Note that when the case is 2 threads 2 child processes the multithreading is faster after that, adding threads and child processes will cause some overhead and it might cause the multiprocessing to be faster. Other reason is that the speed depends on what is running on the CPU at the time in addition to some other conditions. That’s why we should take average execution time when calculating or deciding which approach is better. Theoretically, multithreading is faster since the same data is accessed and no need for any IPC method.

1.2 Amdahl’s law

To find the sequential part, and parallel part percentage I measured the time taken by the naïve as a whole and the time taken by the part, we will make parallel (the frequency counting in my case)

I ran the code three times to take the average result and use it for calculations.

The result for the first time:

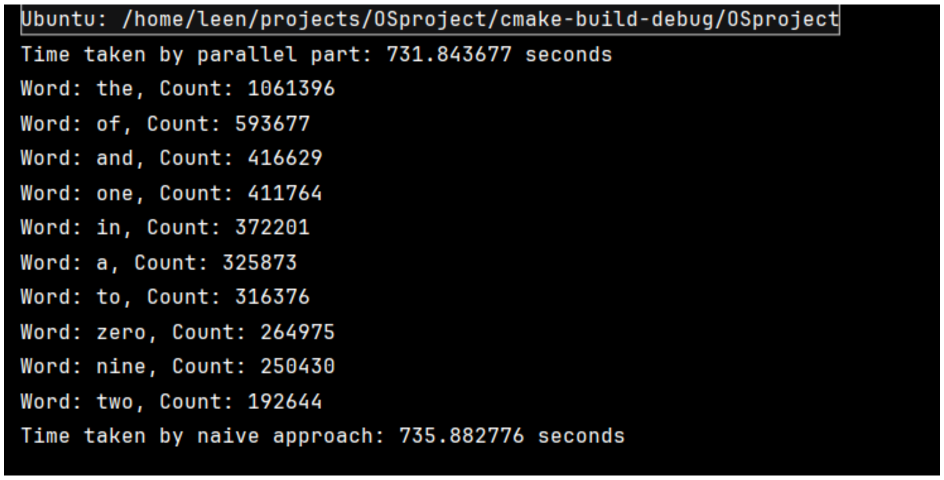


```
Ubuntu: /home/leen/projects/OSproject/cmake-build-debug/OSproject
Time taken by parallel part: 829.051616 seconds
Word: the, Count: 1061396
Word: of, Count: 593677
Word: and, Count: 416629
Word: one, Count: 411764
Word: in, Count: 372201
Word: a, Count: 325873
Word: to, Count: 316376
Word: zero, Count: 264975
Word: nine, Count: 250430
Word: two, Count: 192644
Time taken by naive approach: 831.924090 seconds
```

Time taken by parallel part: 829.051616 seconds

Time taken by naive approach: 831.924090 seconds

The result for the second time:

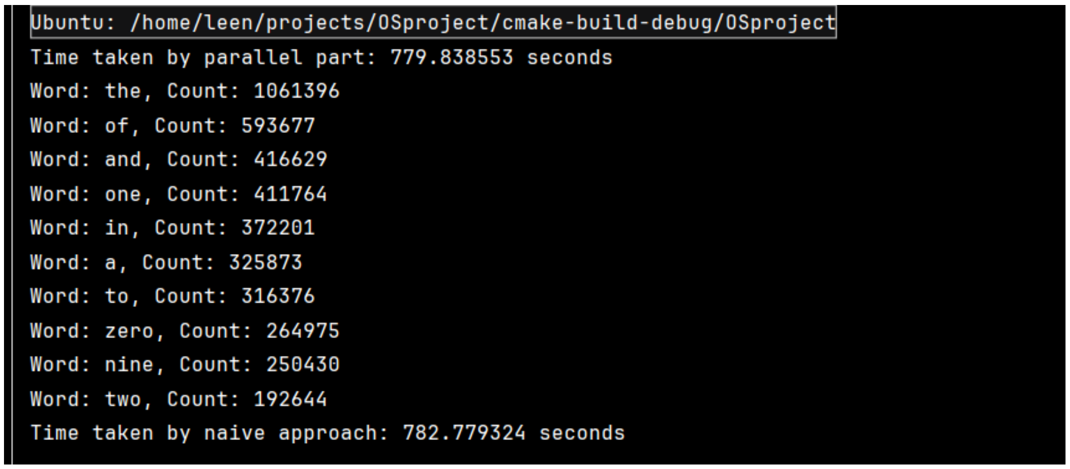


```
Ubuntu: /home/leen/projects/OSproject/cmake-build-debug/OSproject
Time taken by parallel part: 731.843677 seconds
Word: the, Count: 1061396
Word: of, Count: 593677
Word: and, Count: 416629
Word: one, Count: 411764
Word: in, Count: 372201
Word: a, Count: 325873
Word: to, Count: 316376
Word: zero, Count: 264975
Word: nine, Count: 250430
Word: two, Count: 192644
Time taken by naive approach: 735.882776 seconds
```

Time taken by parallel part: 731.843677 seconds

Time taken by naive approach: 735.882776 seconds

The result for the third time:



```
Ubuntu: /home/leen/projects/OSproject/cmake-build-debug/OSproject
Time taken by parallel part: 779.838553 seconds
Word: the, Count: 1061396
Word: of, Count: 593677
Word: and, Count: 416629
Word: one, Count: 411764
Word: in, Count: 372201
Word: a, Count: 325873
Word: to, Count: 316376
Word: zero, Count: 264975
Word: nine, Count: 250430
Word: two, Count: 192644
Time taken by naive approach: 782.779324 seconds
```

Time taken by parallel part: 779.838553 seconds

Time taken by naive approach: 782.779324 seconds

Calculations

Average times:

Parallel:

829.051616 + 731.843677 + 779.838553 /3 = 780.244616 seconds

Naive:

831.924090 +735.882776 +782.779324/ 3 = 783.528063 seconds

Serial percentage:

Serial Time (average)=Naive Time (average)−Parallel Time (average)=783.528063−780.244616=3.283447seconds

Serial Percentage = Serial Time/Naive Time × 100 = 3.283447 /783.528063 × 100 ≈ 0.42 %

parallel percentage:

```
Parallel Percentage=100-Serial Percentage≈99.58%
```

Amdahl’s law:

Given a S=0.0042. the maximum theoretical speedup is:

$$\frac{1}{S+\frac{1-S}{N}}=\frac{1}{0.0042+\frac{0\cdot 9958}{N}}$$

Since I have 8 cores and using the Amdahl’s law shown above

The maximum speedup theoretically is ≈7.77

Optimal number of child processes and threads:

Amdahl's law suggests that the optimal number of processes or threads increases with the number of available cores, but in practice after some point creating more child processes or threads does not improve speedup on the contrary, it can cause some sort of overhead which makes the code even slower.

After running the multiprocessing code for 2,4,6 and 8 processes as provided in the previous part we have found that the best time was achieved with 6 child processes! The execution time for 6 processes was 437 seconds, while 8 processes took 441 seconds—just slightly slower but this shows that after some point adding more processes will not be helpful because of the overhead and even if it’s not slower the difference will not be significant.

After running the multithreading code for 2,4,6 and 8 threads as provided we found that the best time was achieved with 8 threads with an execution time of 428 seconds which is better than the 6 threads one with a good difference 483.

It is important to keep in mind that these numbers can vary depending on the device’s conditions such as the number of processes running on the CPU. However, the main idea remains consistent: using 6 child processes is better than 8 in this case, and 8 threads is the best. the relative concept remains constant regardless of the specific numbers.

Discussion:

2.1 Table of performance compression

The criteria	Naïve code	Multiprocessing code	Multithreading code
Execution time (best case in seconds)	735.88 sec	(6 child processes) 438 sec	(8 threads) 428 sec
Resource usage	Low (single process and thread)	High (multiple processes each has its own pipe)	Average (shared memory but multiple threads)
scalability	Poor	Excellent	Good (depends on the condition)
Complexity	Easy	More complex	More complex

2.2 The difference in performance discussion

1.2.1 Execution time:

After measuring time more than once for each approach the best execution time for the naïve code was 735.88 seconds while the best for the multiprocessing was with 6 child processes with an execution time of 438 sec. finally, the best execution time for multithreading was with 8 threads with a value of 428 sec.

This result is as expected because in the naïve all the code is sequential while in the multiprocessing and multithreading the work is divided and processed in parallel. The multithreading works slightly better since the threads uses the same resources and no time is taken to get the resources.

1.2.2 Resource usage:

The naïve uses the least resources since it runs as a single process in a single thread. The multiprocessing consumes more memory and CPU because more than one process are running at the same time. Multithreading uses shared memory which makes it more efficient than multiprocessing.

A really interesting thing I noticed while working on the project is really how much resources the multiprocessing takes if it was not handled correctly one of the errors, I had caused having temp files with a massive size (100 GB!!) and will not removed unless you remove it manually. After the fixing of the problem the code worked without any problems and taking all of the resources.

1.2.3 Scalability:

Naïve doesn't scale since it runs on a single core. Multiprocessing scales on multicore systems, same for multithreading. This is clearer when we try a small file instead of our large dataset.

1.2.4 complexity:

naïve is straight forward. Multiprocessing needs a setting up for an IPC (pipes in our case). Multithreading needs management for threads and race condition (using mutex in our case)

Conclusion

In this project, we explored three different approaches which are: naïve approach, multiprocessing, and multithreading. we observed the differences between them and the importance of using multiprocessing and multithreading for efficiency. The naïve code is simple easy to implement but it takes a lot more time than the two other approaches specially for large dataset like in our case. The multiprocessing is fast but needs more resources than other approaches as it involves multiple processes, and some type of IPC must be chosen and used, in our project I chose pipes, one for each process. Finally, the multithreading. It is the fastest since the threads uses shared memory. It also uses less resources than multiprocessing so theoretically it is the best. However, we might need to use other approaches in specific cases. We also learned that not always more processes/threads mean better performance. Sometimes more processes/threads mean more overhead and sometimes the difference is not that benefic so its not worth it.