Faculty of Engineering & Technology Electrical & Computer Engineering Department

OPERATING SYSTEMS

ENCS3390

Process and Thread Management

Prepared by: Raghad Jamhour 1220212

Instructor: Dr. Abdel Salam Sayyad

Section: 1

Date: 11/25/2024

Table of Contents

Table of Tables

Abstract:

This project identifies the top 10 most frequent words in the dataset using three approaches: naive (single-threaded), multiprocessing, and multithreading. Execution times are measured for each approach, testing 2, 4, 6, and 8 processes or threads to find the best performance. The results are analyzed using Amdahl’s law to understand the impact of parallelization and identify the optimal number of threads or processes.

Implemented Approaches:

Multithreading Approach

The multithreading approach divides the input file into smaller chunks, which are processed in parallel using multiple threads. The file is read and split into chunks with fseek() and fread(), and each chunk is passed to a separate thread for processing. The threads perform word tokenization and frequency counting using an AVL tree then insert the words with their frequencies into an array of structs to sort it and get top words from each thread. After each thread completes its task, the results are sorted then merged safely into a global array (Top 100 words of each thread) using a mutex to avoid data races. The frequent words are then sorted using qsort() and printed.

Libraries and APIs Used:

POSIX Threads (pthread): This library is central to implementing multithreading in the approach. The key functions from this library are: pthread_create() which was used to create threads that process chunks of the file concurrently. pthread_mutex_lock() and pthread_mutex_unlock() which ensured thread safety when merging results from different threads. Finally, pthread_join() which ensures the main thread waits for all worker threads to finish before proceeding with the result merging.

Standard I/O (stdio.h): The fseek() and fread() functions from this library are used for reading the input file and splitting it into smaller chunks for parallel processing.

C Standard Library (stdlib.h): The qsort() function from this library is used to sort the word-frequency pairs in descending order of frequency using Quick sort algorithm.

Multiprocessing Approach

The multiprocessing approach uses separate processes instead of threads. The input file is split into chunks, with each chunk processed by a different child process. The child processes are created using fork(), which creates a copy of the parent's memory, meaning any changes made by the child are not reflected in the parent, as they each have separate memory spaces. After processing using the same logic as the Multithreading (AVL trees and an array of structs), each child writes its results to a temporary file so the parent can see the changes done by the child. This approach is followed instead of using shared memory or pipes for simplicity. The parent process reads these temporary files, merges the results, and then sorts and prints the top frequent words. This approach ensures complete isolation between processes and utilizes multiple CPU cores effectively.

Libraries and APIs Used

Standard I/O (stdio.h): Functions such as fseek() and fread() are used to read the file and split it into chunks. fprintf() is used to store the results in temporary files.

System Calls (unistd.h): fork() which creates child processes for parallel execution. Each child process works independently on a chunk of the file. waitpid() where the parent process waits for all child processes to finish before merging and printing the results.

C Standard Library (stdlib.h): As in the multithreading approach, qsort() is used to sort the arrays based on frequency.

Important Functions Implemented:

insert(): Inserts words into an AVL tree, incrementing their frequency if already present. This function is used in both multithreading and multiprocessing approaches to efficiently maintain word counts during the processing phase.

traverseInOrder(): Traverses the AVL tree and stores words and their frequencies in an array. This function is essential for gathering all the word-frequency pairs after insertion, ready for further processing in both threading and multiprocessing.

printTopWords(): In multiprocessing, this function displays the top 10 most frequent words in the array. In multithreading, it inserts the top 100 words from each thread into a global array using a mutex lock, ensuring that threads efficiently share their results.

mergeResults(): Combines the word frequencies from each child process or thread. This function iterates over the global array, merging results from each concurrent execution, and then prints the top 10 most frequent words to provide a final summary of the data.

writeTopWordsToFile(): In multiprocessing, this function writes the top words and their frequencies from each child process to a temporary file for further processing. This allows for easy integration and merging of results across different processes.

readWordsFromFile(): Reads word-frequency from the temporary files generated by each process or thread. This function merges the data into a final array, ensuring that all results are accurately combined before being sorted and printed.

threadFunc(): A function used in multithreading, where each thread processes a portion of the input, inserts words into the AVL tree, and stores the results in an array. This function is responsible for dividing the task between threads, and ensuring that each thread insert its most frequent words into a global array.

These functions collectively optimize the processing of words, ensuring the program can handle large datasets with high performance and accuracy.

Environment:

Cores: The system, a Lenovo ThinkPad T450s, is equipped with 2 physical cores (Intel Core i5-5300U, with 2 threads per core, resulting in a total of 4 logical cores).

Speed: The CPU operates at a base clock speed of 2.30 GHz, with a turbo boost capability of up to 2.90 GHz.

Memory: The system has 14 GB of RAM, with 3.4 GB currently in use and 10 GB available.

Operating System: The operating system in use is Ubuntu 24.10 (Oracular), a Linux-based distribution.

Programming Language: The project was developed using the C programming language for both sequential and parallel tasks.

IDE Tool: Code::Blocks was used as the Integrated Development Environment (IDE) for coding, compiling, and debugging the C program.

Virtual Machine: The project was executed directly on a physical machine, and no virtual machine was used.

Analysis According to Amdahl's law

1. **Serial Part of the Code**
   To calculate the **serial part** of the code, we first need to estimate the **parallel fraction (P)**, which is the portion of the program that can be parallelized. The serial part is the complement of that, calculated as: Serial Fraction = 1 - P

   To calculate the parallel fraction P from the data provided, we use Amdahl's Law:

   Speedup = Tnaive / Tparallel (calculated from running the program)

   Speedup = 1 / ((1 - P) + (P / N))

   Where Tnaive: Execution time for the naive approach (serial execution).
   Tparallel: Execution time for the parallel approach.
   N: Number of CPUs.

   2 Threads:
   Tparallel = 5.576 seconds
   Speedup = 9.427 / 5.576 = 1.690
   P = (4 * (1.690 - 1)) / (1.690 * (4 - 1)) = (4 * 0.690) / (1.690 * 3) = 0.544

   4 Threads:
   Tparallel = 4.422
   Speedup = 9.427 / 4.422 = 2.132
   P = (4 * (2.132 - 1)) / (2.132 * (4 - 1)) = (4 * 1.132) / (2.132 * 3) = 0.7073

   6 Threads (with 4 cores):
   Tparallel = 4.279
   Speedup = 9.427 / 4.279 = 2.203
   P = (4 * (2.203 - 1)) / (2.203 * (4 - 1)) = (4 * 1.203) / (2.203 * 3) = 0.727

   8 Threads (with 4 cores):
   Tparallel = 4.445
   Speedup = 9.427 / 4.445 = 2.121
   P = (4 * (2.121 - 1)) / (2.121 * (4 - 1)) = (4 * 1.121) / (2.121 * 3) = 0.705

   Now, we calculate the serial fraction as:

   2 Threads: S = 1 - 0.544 = 0.455

   4 Threads: S= 1 - 0.7073 = 0.2927

   6 Threads: S = 1 - 0.727 = 0.273

   8 Threads: S = 1 - 0.705 = 0.295

The serial portion ranges from 18.34% (for 2 threads) to 29.5% (for 8 threads).

As the number of threads exceeds the number of available cores, performance will not continue to scale linearly. After 4 threads, any additional threads would result in reducing performance due to the overhead of context switching and managing more threads than the available cores. Thus, theoretically 4 threads would be the optimal number to achieve the best performance on a 4-core system. Note that these values closely align with values obtained from Multiprocessing since they are quite close.

2. **Maximum Speedup According to the Available Number of Cores**

Amdahl's Law tells us the maximum speedup we can achieve is limited by the serial portion of the program. The formula for the maximum speedup is:

Speedup = 1 / ((1 - P) + (P / N))

Given the serial fractions calculated earlier, we can calculate the maximum speed up

For 2 Threads (P = 0.544):
Speedup = 1 / (0.455 + (0.544 / 4))
= 1 / (0.455 + 0.136)
= 1 / 0.591 = 1.692

For 4 Threads (P = 0.7073):
Speedup = 1 / (0.2927 + (0.7073 / 4))
= 1 / (0.2927 + 0.1768)
= 1 / 0.4695= 2.13

For 6 Threads (P = 0.727):
Speedup = 1 / (0.273 + (0.727 / 4))
= 1 / (0.273 + 0.1818)
= 1 / 0.4548 = 2.20

For 8 Threads (P = 0.705):
Speedup = 1 / (0.295 + (0.705 / 4))
= 1 / (0.295 + 0.1763)
= 1 / 0.4713 = 2.12

Maximum speed up is achieved using 6 threads since it has the highest value of 2.2

3. **Optimal Number**

Optimal Number: Based on the data, the optimal number of threads is 6, as it yields the highest speedup. Same calculations are done for Multiprocessing resulting in 4 processes having the highest speed up (2.25 which is close to the 6 threads). According to Amdahl's law the optimal number of threads/processes would be 4 Since the system has only 4 cores, using 4 is more practical for the best balance of performance and resource usage.

Comparison of Execution Times for Naive, Threaded, and Multiprocessing Approaches

Approach	Execution Times (Seconds)	Average Time (Seconds)
Naive Execution	9.636, 9.285, 9.223, 9.268, 10.002, 9.35	9.427
2 Threads	5.968, 5.537, 5.178, 5.618, 5.582, 5.523	5.576
4 Threads	4.672, 4.295, 4.492, 4.403, 4.283, 4.385	4.422
6 Threads	4.207, 4.246, 4.259, 4.34, 4.411, 4.331	4.279
8 Threads	4.5883, 4.392, 4.092, 4.554, 4.35, 4.395	4.445
2 Processes	5.386, 5.661, 5.189, 5.065, 5.483, 5.164	5.325
4 Processes	4.287, 4.183, 4.145, 4.122, 4.199, 4.182	4.186
6 Processes	4.631, 4.285, 4.34, 4.3, 4.22, 4.379	4.359
8 Processes	4.475, 4.302, 4.299, 4.43, 4.2, 4.218	4.321

Table 1: Comparison of Execution Times

Comments on the Differences in Performance:

Naive Execution:

The naive approach has the highest execution time at 9.427 seconds, which is expected since no parallelism is used.

Threaded Execution:

2 Threads: Execution time of 5.576 seconds, showing a significant improvement over the naive approach.

4 Threads: Execution time of 4.422 seconds, further reducing the time compared to 2 threads, showing continued improvement in performance.

6 Threads: Execution time of 4.279 seconds, slightly better than 4 threads. Which makes it the optimal number in Multithreading.

8 Threads: Execution time of 4.445 seconds, slightly worse than 6 threads, indicating that beyond a certain point, the overhead and thread management might reduce the performance of adding more threads.

Multiprocessing Execution:

2 Processes: Execution time of 5.325 seconds, showing performance similar to that of 2 threads.

4 Processes: Execution time of 4.186 seconds, offering better performance than 2 processes, and outperforming 4 threads slightly. This shows that multiprocessing performs better with more processes. This is the best result in the multiprocessing approach.

6 Processes: Execution time of 4.359 seconds, slightly worse than 4 processes. This is likely due to the number of processes exceeding the number of CPU cores, which introduces overhead from context switching.

8 Processes: Execution time of 4.321 seconds. Note that performance did not improve compared to 4 processes because the number of processes exceeds the CPU cores, leading to context switching overhead.

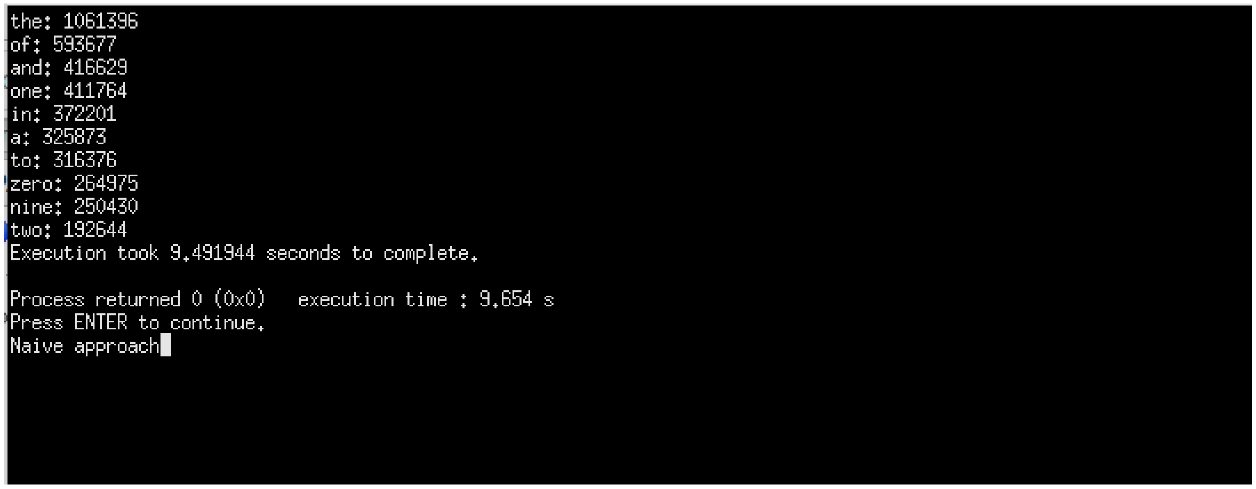
Conclusion

Parallel execution using either threads or processes significantly reduces the execution time compared to the naive approach. The naive execution time of 9.427 seconds is reduced to around 4.186 - 5.576 seconds with parallel approaches, showing a clear improvement.

Threading vs. Multiprocessing: The performance of threading and multiprocessing is quite similar. 6 threads (4.279 seconds) outperform 6 processes (4.359 seconds) which is normal since the processes depend on the number of cores which is exceeded is this case causing it to slow down due to overhead. However, the differences are low, suggesting that for this workload, both parallelization methods are effective.

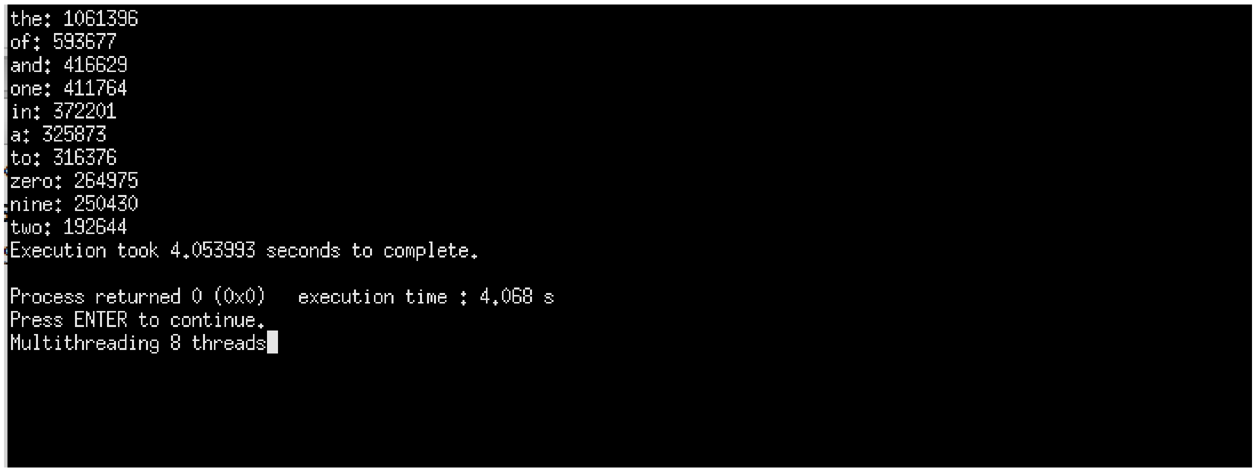
Optimal Configuration: The best performance is achieved with 6-thread configuration (4.279 seconds) and with 4 processes (4.186 seconds). The results suggest that increasing the number of threads or processes further could lead to reducing the performance due to the overhead from switching. Therefore, choosing the optimal number of threads/processes is crucial to avoid inefficiency.

Appendix



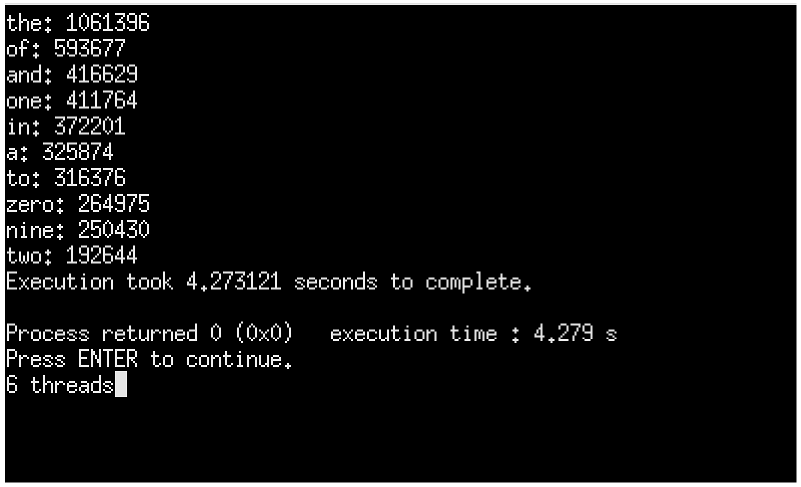
```
the: 1061396
of: 593677
and: 416629
one: 411764
in: 372201
a: 325873
to: 316376
zero: 264975
nine: 250430
two: 192644
Execution took 9.491944 seconds to complete.

Process returned 0 (0x0)   execution time : 9.654 s
Press ENTER to continue.
Naive approach
```



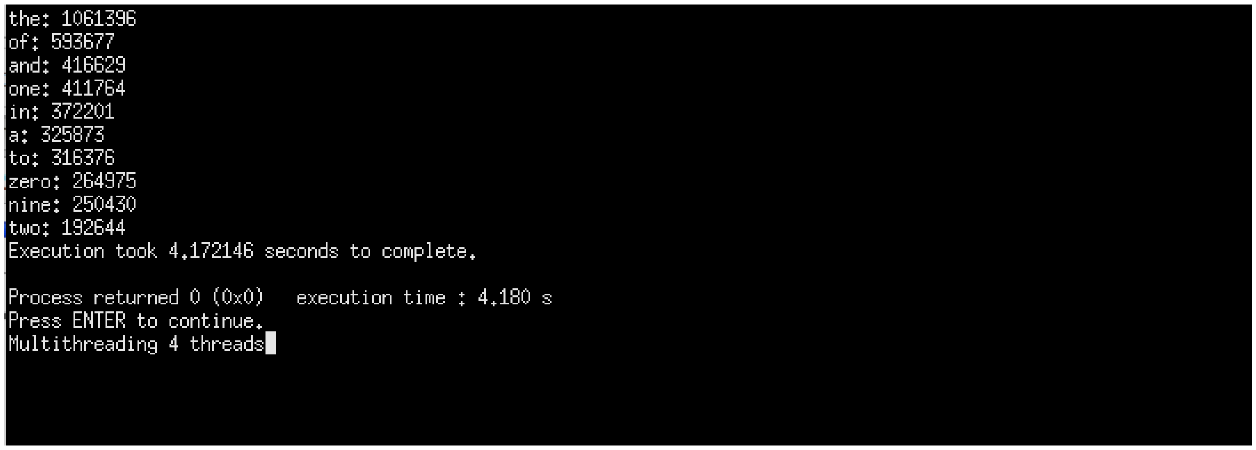
```
the: 1061396
of: 593677
and: 416629
one: 411764
in: 372201
a: 325873
to: 316376
zero: 264975
nine: 250430
two: 192644
Execution took 4.053993 seconds to complete.

Process returned 0 (0x0)   execution time : 4.068 s
Press ENTER to continue.
Multithreading 8 threads
```



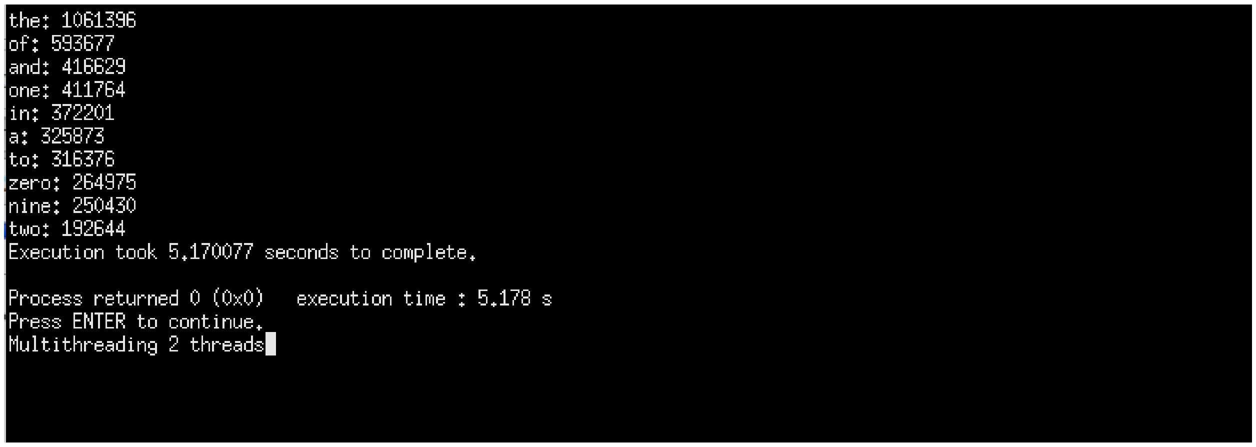
```
the: 1061396
of: 593677
and: 416629
one: 411764
in: 372201
a: 325874
to: 316376
zero: 264975
nine: 250430
two: 192644
Execution took 4.273121 seconds to complete.

Process returned 0 (0x0)   execution time : 4.279 s
Press ENTER to continue.
6 threads
```



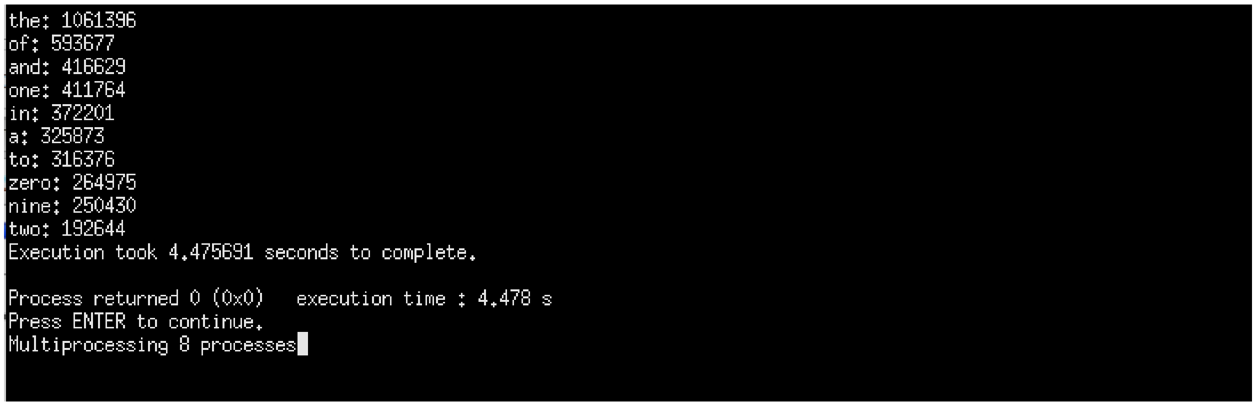
```
the: 1061396
of: 593677
and: 416629
one: 411764
in: 372201
a: 325873
to: 316376
zero: 264975
nine: 250430
two: 192644
Execution took 4.172146 seconds to complete.

Process returned 0 (0x0)   execution time : 4.180 s
Press ENTER to continue.
Multithreading 4 threads
```



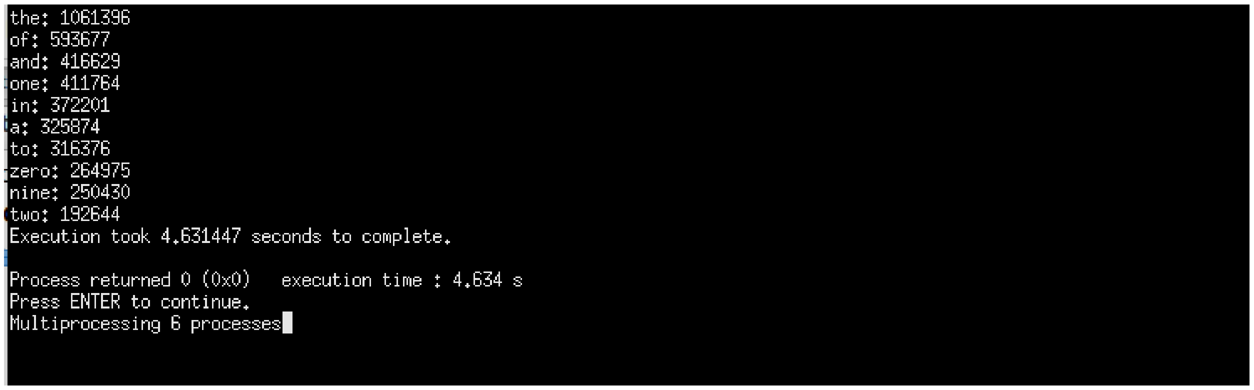
```
the: 1061396
of: 593677
and: 416629
one: 411764
in: 372201
a: 325873
to: 316376
zero: 264975
nine: 250430
two: 192644
Execution took 5.170077 seconds to complete.

Process returned 0 (0x0)   execution time : 5.178 s
Press ENTER to continue.
Multithreading 2 threads
```



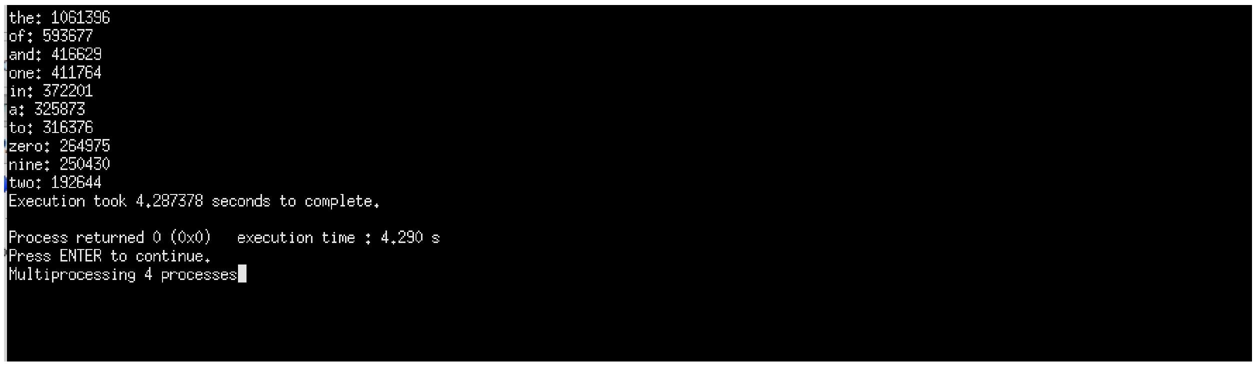
```
the: 1061396
of: 593677
and: 416629
one: 411764
in: 372201
a: 325873
to: 316376
zero: 264975
nine: 250430
two: 192644
Execution took 4.475691 seconds to complete.

Process returned 0 (0x0)   execution time : 4.478 s
Press ENTER to continue.
Multiprocessing 8 processes
```



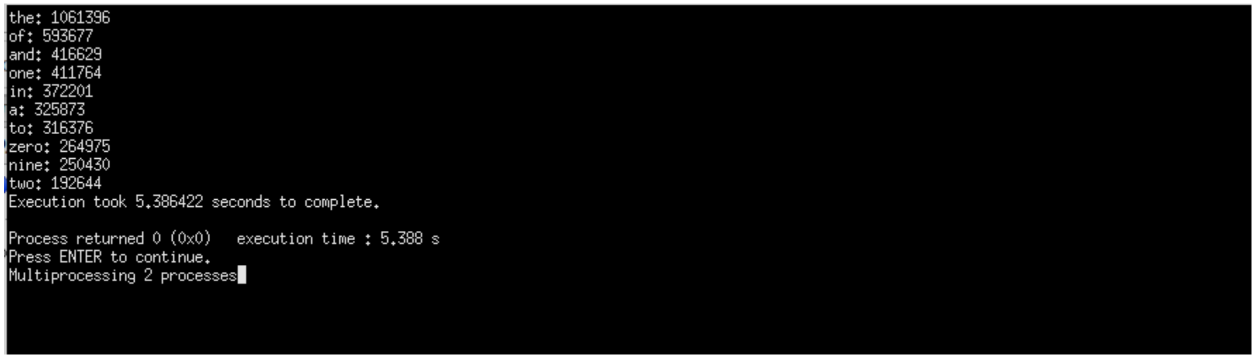
```
the: 1061396
of: 593677
and: 416629
one: 411764
in: 372201
a: 325874
to: 316376
zero: 264975
nine: 250430
two: 192644
Execution took 4.631447 seconds to complete.

Process returned 0 (0x0)   execution time : 4.634 s
Press ENTER to continue.
Multiprocessing 6 processes
```



```
the: 1061396
of: 593677
and: 416629
one: 411764
in: 372201
a: 325873
to: 316376
zero: 264975
nine: 250430
two: 192644
Execution took 4.287378 seconds to complete.

Process returned 0 (0x0)   execution time : 4.290 s
Press ENTER to continue.
Multiprocessing 4 processes
```



```
the: 1061396
of: 593677
and: 416629
one: 411764
in: 372201
a: 325873
to: 316376
zero: 264975
nine: 250430
two: 192644
Execution took 5.386422 seconds to complete.

Process returned 0 (0x0)   execution time : 5.388 s
Press ENTER to continue.
Multiprocessing 2 processes
```