FACULTY OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

ENCS3390: Operating System Concepts

PROJECT #1 Report

Prepared by: Malak Milhem 1220031

Instructor: Dr. Yazan Abo Farha

Sec. 1

Dec.1st. 2024

Abstract:

In this project, I compared three approaches—naïve, multithreading, and multiprocessing—for counting the most frequent words in the *enwik8* dataset. The naïve approach processes the entire file sequentially using one thread, taking the longest time of 13.3 minutes. For the multithreading approach, I used POSIX threads, with 6 threads reducing the execution time to 8.089 minutes, achieving a speedup of 1.645x. The multiprocessing approach, which uses separate processes to divide the task, performed the best, reducing the time to 7.975 minutes and achieving a speedup of 1.67x. While multiprocessing eliminates the need for synchronization, it introduces overhead during result merging.

Using Amdahl's Law, I found that 52% of the code is serial, limiting the potential speedup from more threads or processes. Both multithreading and multiprocessing reached their optimal performance with 6 threads or processes, after which further improvements were minimal. These results show the trade-offs between parallelism, memory usage, and synchronization overhead. Multiprocessing provided the best performance overall, but multithreading could still be a better option in situations where memory usage is a concern.

Table of Contents

List of tables:

Theory:

To explain the three primary approaches used to analyze the file in our project, first the naïve approach, which processes tasks sequentially without any parallelism, the entire dataset is read and processed, and word frequencies are counted and ordered in a single flow of execution, one after the other. This approach is simple to implement and suitable for small-scale, simple tasks because it uses only one core, but on the other hand, usage of only one core makes it inefficient and slow for large datasets.

Multithreading approach, in this approach multiple threads within a single process work concurrently. Each thread takes a portion of the dataset to process, and they share the same memory space, which allows for efficient data sharing. Multithreading speeds up execution on multi-core processors by utilizing several threads to perform tasks at once. However, careful synchronization is needed to avoid issues like race conditions.

Lastly, the multiprocessing approach, it creates multiple independent processes, each with its own memory space. Each process runs in parallel on a separate core, allowing true parallelism. Since processes do not share memory, there is no risk of race conditions, but the approach comes with higher memory usage and communication overhead between processes. Despite this, multiprocessing fully utilizes multi-core CPUs and is well-suited for large datasets tasks.

The environment’s Description:

Computer Specifications:

Processor: Intel Core i7-10510U CPU @ 1.80GHz

Number of Cores: 4 cores

Logical processors: 8

Base Speed: 1.80 GHz

Memory (RAM): 8 GB

Operating System: Windows 10 Pro, Ubuntu 22.04.5 LTS

Programming Language: C

IDE Tool: Visual Studio Code, and I installed WSL 2

I didn’t use a Virtual Machine.

Procedure:

Multithreading:

In my code I used the POSIX threads library <pthread.h> to achieve multithreading,

Pthread_t represents a thread identifier, and `pthread_t *threads;` is used to store IDs of all created threads.

```
pthread_create(&threads[i], NULL, threadscount, &ThreadsInfo[i]);
```

the function creates a new thread, first parameter is a pointer to store the thread ID, the third is a function to execute in the new thread, and the fourth is a pointer to the data structure passed to the thread as an argument.

Thread Data structure:

```
typedef struct {

   char *startofpart;   // start position in the file part
   long partsize;       // size of the file part
   WordFreq *localList;  // local frequency list
   int threadCount;      // local word count
} Threadstruct;
```

```
pthread_join(threads[i], NULL);
```

this function waits for a thread to complete its execution, the first parameter is the ID of the thread to wait for. This function is important to ensure the main program waits until all threads finish processing their assigned tasks.

Now for thread synchronization to prevent race conditions when threads access shared resources, we used mutexes,

```
pthread_mutex_init(&mergeMutex,NULL);
```

the purpose of this function is to initialize a mutex for protecting shared resources during list merging, where &mergeMutex is a pointer to the mutex

```
pthread_mutex_lock(&mergeMutex);
```

this line locks a mutex to ensure only one thread accesses the critical section, so we prevent concurrent access to the global list while merging thread-local lists into it.

```
pthread_mutex_unlock(&mergeMutex);
```

to unlock a mutex to allow other threads to enter the critical section which is merging.

```
pthread_mutex_destroy(&mergeMutex);
```

at the end of our main we destroy the mutex after all threads finish.

In our thread function (void *threadscount(void *arg){) the thread processes a portion of the file as divided (from startofpart till partsize) to count word frequencies and update the local frequency list (localList) then it calls (mergeLists()) to merge the local list into the global list.

Lets point out file handling APIs used, fopen(), fseek(), and ftell() are used to open the file, set the file pointer, and determine the file size, and fclose() closes the file after processing.

And for memory allocation, malloc() and calloc() allocate memory dynamically for thread data structures (ThreadsInfo, localList, and startofpart).

Key functions include findWord**()**, which searches for a word in the word list, and insertWord**()**, which inserts a word into the list, updating its frequency if it already exists. And compareWordFreq**()** is used as a comparison function for qsort() to sort the global list of words in descending order of frequency. And lastly, we used gettimeofday() from the library <sys/time.h> to measure the execution time of the word frequency counting and sorting process from (start) to (end).

By applying a multithreading approach we reduced total execution time due to threads processing different parts of the file in parallel.

Multiprocessing:

in the multi threading approach we used several system calls, standard library functions, and POSIX APIs. Process management is achieved using fork() to create child processes, while waitpid() ensures the parent process waits for all child processes to finish execution. File handling is performed using fopen(), fread(), fclose(), ftell(), and fseek() to read and divide the file. To store word frequencies, the program uses dynamic memory allocation with malloc() and free(). Finally, sorting and time measurement are achieved with qsort() to sort words by frequency and we used gettimeofday() to measure execution time for performance analysis.

The program follows a structured approach to divide, process, and merge results. First, the parent process reads the file, determines its size using ftell(), and splits it into chunks, one for each child process. The fork() system call is used to create multiple child processes, each of which processes its assigned portion of the file. The file is divided into chunks such that word boundaries are not split. Each child process maintains its own local list of words and their frequencies. The local results are written to separate files (process_1.txt, process_2.txt, etc.), it is defined in the processPart**()** function which processes a portion of the file, extracting words and updating the word list. After all child processes finish execution, the parent process reads these results and merges them into a global word list. The word list is sorted using qsort() to identify the top 10 most frequent words. The execution time is measured using gettimeofday().

The multiprocessing approach successfully reduced execution. it efficiently divided the workload among multiple processes, leveraging the power of parallelism to achieve a significant speedup.

Analysis:

Approach	Naïve	Multithreading				Multiprocessing			
Num of threads/ processes	---	2	4	6	8	2	4	6	8
Execution time (#1)	13.7min	10.2 min	8.399 min	8.44 min	8.15 min	9.8 min	9.19 min	7.85 min	7.9666 min
Execution time (#2)	14.27 min	10.29 min	8.397 min	7.738 min	8.16 min	10.6 min	9.227 min	8.1 min	8.178 min
Execution Time (#3)	12.18 min	10.37 min	---	---	---	---	---	---	---
Avarege time	13.3 min	10.28 min	8.398 min	8.089 min	8.155 min	10.2 min	9.2 min	7.975 min	8.07 min

Table 1 : execution time of all approaches

Amdahl's Law defines the speedup of a system when part of it is parallelized.

$S = \frac{1}{(1-P)\ P/n}$

Where

p = ratio of the code that is parallelizable

(1−p) = ratio of the code that is serial

n = number of processes or threads

from table 1, the serial portion, the execution time when done with naïve approach, = 13.3 min

for multithreading, the parallel portion, for 2 threads = 10.28 min

so the effective speed up S will be 13.3 / 10.28 = 1.29

now for n = 2 threads, $S = \frac{1}{(1-P)\ P/n}$ Rearranging to solve for (1−p): 1.29 = $\frac{1}{(1-P)+p/2}$

after solving p = 0.449, the serial portion (1 – p) = 0.551

now for n = 6 threads, $S = \frac{1}{(1-P)\ P/n}$ Rearranging to solve for (1−p): 1.667 = $\frac{1}{(1-P)+p/6}$

after solving p = 0.48, the serial portion (1 – p) = 0.52

Thus, the serial portion of the code is approximately 52% and the parallel portion is 48% for the multithreading approach with 6 threads.

In the same steps we can find the speedup for the other thread and process counts :

For multithreading:

- **2 threads**: 10.28 min, speedup S2 ≈ 1.29
- **4 threads**: 8.398 min, speedup S4=13.3/8.398 ≈ 1.58
- **6 threads**: 8.089 min, speedup S6=13.3/8.089 ≈1.645
- **8 threads**: 8.155 min, speedup S8=13.3/8.155 ≈1.63

For multiprocessing:

- **2 processes**: 10.2 min, speedup S2=13.3/10.2 ≈1.30
- **4 processes**: 9.2 min, speedup S4=13.3/9.2 ≈1.45
- **6 processes**: 7.975 min, speedup S6=13.3/7.975 ≈1.67
- **8 processes**: 8.07 min, speedup S8=13.3/8.07 ≈1.65

Optimal Number of Threads/Processes:

- The execution time improves significantly from 2 threads to 6 threads, but there is minimal improvement from 6 to 8 threads.
- The same is seen for processes, where 6 processes yield the fastest execution time of 7.975 min and max speed up of S = 1.67**.**
- Hence, the optimal number of threads or processes is 6, as increasing to 8 threads or processes provides minimal additional speedup.

The Differences in Performance

The performance of the three approaches—naïve, multithreading, and multiprocessing—demonstrates the critical role of parallelism in reducing execution time, optimizing resource usage, and enhancing speedup. Each approach has distinct characteristics in terms of execution time, memory usage, synchronization overhead, and CPU utilization.

The naïve approach operates sequentially, processing the file from start to finish using a single core. This simplicity comes at a cost: the longest execution time, averaging 13.3 minutes. Since only one core is utilized, the approach is unable to leverage the capabilities of modern multi-core processors.

In contrast, the multithreading approach introduces parallelism by creating multiple threads that process different parts of the file simultaneously. Each thread operates within the same memory space, which facilitates efficient data sharing. However, this also necessitates the use of mutex locks to prevent race conditions when threads access shared resources like the global word list. Synchronization overhead from mutex locks increases as the number of threads rises, limiting performance gains. While execution time improves with additional threads, the optimal number of threads is found to be 6, yielding an average execution time of 8.089 minutes. Beyond 6 threads, minimal performance improvements are observed due to overhead from synchronization and context switching on the CPU.

The multiprocessing approach delivers the best performance among the three. Unlike threads, each process has its own memory space, thereby eliminating the need for mutex locks. Child processes operate independently, and their results are later merged by the parent process. This isolation avoids race conditions but introduces additional overhead from file I/O operations required to merge the child processes' results. Nevertheless, multiprocessing achieves the shortest execution time, averaging 7.975 minutes with 6 processes, and provides a speedup of 1.67x compared to the naïve approach. Similar to the multithreading approach, increasing the number of processes beyond 6 provides no significant improvement due to the overhead of file I/O and the saturation of available CPU cores.

The analysis of speedup using Amdahl’s Law reveals the impact of the serial portion of the code on the overall performance. As Amdahl's Law predicts, increasing the number of threads or processes does not yield proportional speedup if a significant portion of the code remains serial. For the multithreading approach, speedup increases from 1.29x with 2 threads to 1.645x with **6** threads, after which it plateaus. Similarly, for the multiprocessing approach, speedup reaches a maximum of 1.67x with 6 processes, and increasing to 8 processes provides negligible additional benefit.

Another key performance factor is memory usage. The naïve approach uses minimal memory, with storage only required for the file, the frequency list, and essential variables. The multithreading approach requires shared memory for threads, along with memory for thread-specific frequency lists. Synchronization through mutex locks adds computational overhead, which grows as more threads are introduced. The multiprocessing approach uses significantly more memory since each process maintains its own memory space. While this avoids shared-memory conflicts, it increases total memory usage and introduces file I/O operations to combine child process results. This increase in file I/O overhead explains why multiprocessing does not achieve substantially faster execution times compared to multithreading, despite its superior isolation of processes.

Conclusion

The analysis of the naïve, multithreading, and multiprocessing approaches highlights the importance of parallelism in reducing execution time and optimizing resource utilization. The naïve approach, while simple, is inefficient for large datasets, as it runs on a single core and takes an average of 13.3 minutes to complete. It serves as a reference point for evaluating the performance of multithreading and multiprocessing.

The multithreading approach improves performance significantly, as it allows multiple threads to process portions of the file concurrently. Using 6 threads reduces the execution time to 8.089 minutes, achieving a speedup of 1.645x. However, increasing the number of threads beyond 6 shows diminishing returns due to mutex overhead and context switching. Synchronization is essential to prevent race conditions, but it also increases computational overhead.

The multiprocessing approach provides the best performance, reducing execution time to 7.975 minutes using 6 processes and achieving a speedup of 1.67x. The key advantage of multiprocessing is the isolation of memory spaces for each process, which eliminates the need for synchronization via mutexes. Instead, child processes operate independently, and their results are merged by the parent process. However, file I/O overhead is introduced during the merging process.

The analysis of Amdahl’s Law reveals that the serial portion of the code is significant, accounting for 52% of the total execution time. This explains why increasing the number of threads or processes beyond **6** does not result in substantial performance gains. Both multithreading and multiprocessing achieve their optimal performance at **6** threads or 6 processes, as increasing this number only introduces more synchronization overhead (in multithreading) or file I/O overhead (in multiprocessing).

In conclusion, the optimal configuration for the user's system (an Intel Core i7-10510U with 4 physical cores and 8 logical processors) is to use 6 threads or 6 processes. The multiprocessing approach is preferred when true parallelism is required, as it avoids shared memory conflicts. However, if memory constraints exist, the multithreading approach may be preferred, as it consumes less memory and avoids the overhead of inter-process communication. This project

highlights the strengths and trade-offs of different parallel computing strategies, illustrating the importance of balancing synchronization overhead, memory usage, and CPU utilization for optimal performance.

Naïve approach output and execution time:

```
unix_malak@DESKTOP-DMC4UQD:~/os/projects/project1$ ./naivewithList
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 to count and find the most frequent words: 730.80313 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 to count and find the most frequent words: 856.42650 sec
unix_malak@DESKTOP-DMC4UQD:~/os/projects/project1$
```

```
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 to count and find the most frequent words: 823.97060 sec
```

Multithreading approach output and execution time:

2 threads:

```
Top 10 most frequent words:
the: 1061396
the: 1061396
of: 593677
of: 593677
and: 416629
one: 411764
in: 372201
a: 325873
to: 316376
zero: 264975
nine: 250430
two: 192644
Time taken : 622.38012 sec
```

```
unix_malak@DESKTOP-DMC4UQD:~/os
Enter num of threads wanted: 2
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 : 617.59406 sec
```

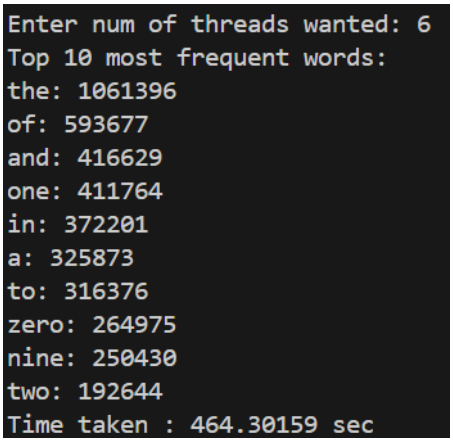
```
unix_malak@DESKTOP-DMC4UQD:~/os
Enter num of threads wanted: 2
Top 10 most frequent words:
the: 1061396
of: 593677
and: 416629
one: 411764
in: 372201
a: 325873
to: 316376
zero: 264975
nine: 250430
two: 192644
Time taken : 612.12101 sec
```

4 threads :

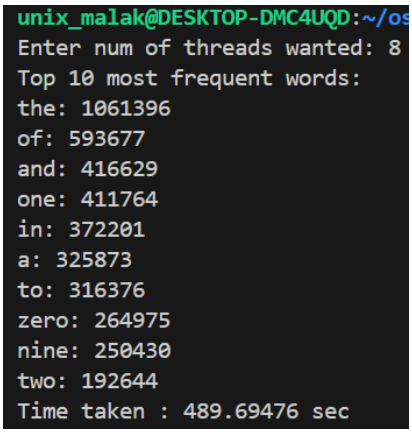
```
unix_malak@DESKTOP-DMC4UQD:~/os
Enter num of threads wanted: 4
Top 10 most frequent words:
the: 1061396
of: 593677
and: 416629
one: 411764
in: 372201
a: 325873
to: 316376
zero: 264975
nine: 250430
two: 192644
Time taken : 503.82241 sec
```

```
Enter num of threads wanted: 4
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 : 503.94898 sec
```

6 threads :

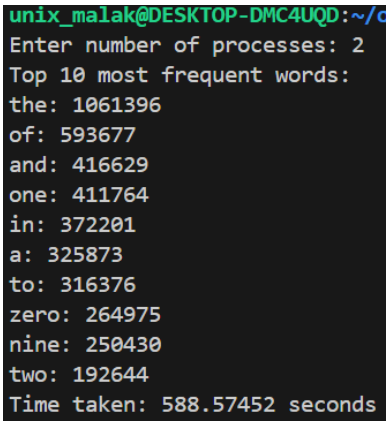


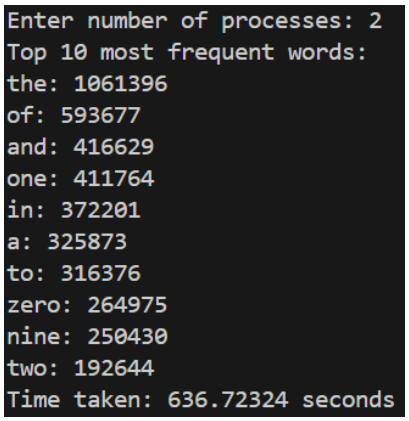
8 threads :



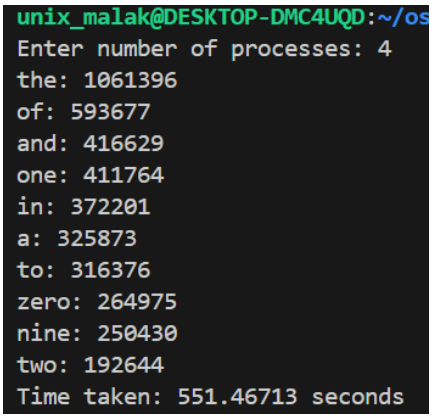
Multiprocessing approach output and execution time:

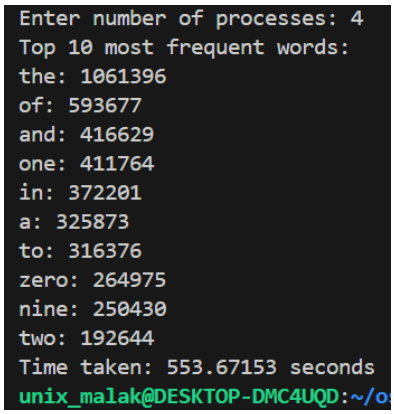
2 processes:



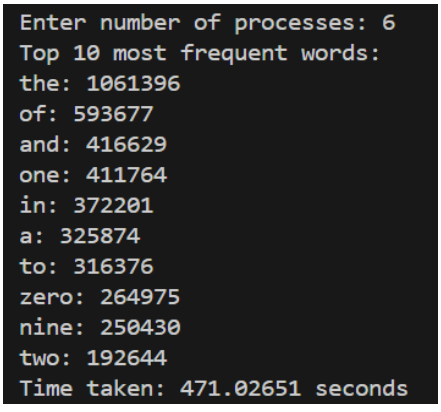


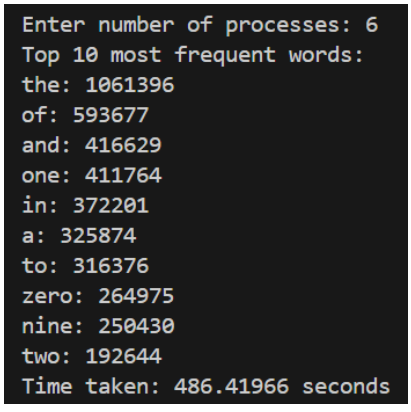
4 processes:





6 processes:





8 processes:

```
Enter number of processes: 8
Top 10 most frequent words:
the: 1061396
of: 593677
and: 416629
one: 411764
in: 372201
a: 325874
to: 316376
zero: 264975
nine: 250430
two: 192644
Time taken: 478.01870 seconds
```

```
Enter number of processes: 8
Top 10 most frequent words:
the: 1061396
of: 593677
and: 416629
one: 411764
in: 372201
a: 325874
to: 316376
zero: 264975
nine: 250430
two: 192644
Time taken: 490.74242 seconds
```