**CMPSC 472 – Project #1**

**File Processing System with Multiprocessing and Multithreading**

**Project Description:**

~ This project is a file processing system that compares the performance of **multithreading** and **multiprocessing** in counting word occurrences across multiple large text files. It allows a user to input a word, then processes seven files in parallel using both techniques. In the multithreading mode, each file is divided into chunks, and multiple threads are created to count occurrences of the word. In the multiprocessing mode, separate processes are spawned for each file, with each process utilizing threads to handle chunks of the file. The system measures and compares the time taken, CPU usage, and memory usage for both approaches, providing insights into their performance efficiency.

**Code Structure:**

* Diagram:
* Implementation Description:

**Program Instructions:**

1. **Compile the Program** – In whatever terminal or IDE you are using, make use of the “gcc” compiler to compile the program.
   1. EX: gcc -o file\_processor file\_processor.c -lpthread
2. **Prepare Input Files** – Ensure that the program has access to the seven text files necessary to be parsed and that the variable “NUM\_FILES” is set accordingly (to seven). These files should all be in the same directory as the program or provide the correct path inside the program code. The default file names consist of:
   1. "bib.txt", "paper1.txt", "paper2.txt", "progc.txt", "progl.txt", "progp.txt", "trans.txt"
3. **Run the Program** – After successfully compiling the program, run the program in your terminal/IDE with the “./” command.
   1. EX: ./file\_processor
4. **Enter the Word to Count** – When prompted by the program, type in the word you would like to search for in the given files and press Enter.
5. **View the Results** – The program should finally display the number of occurrences of the word in each file, the total time taken (in ms) for the search, the CPU usage (in ms), the memory usage (in kb) and the total occurrences of the word across all files. Repeat from step 3 to search for another word.

**Performance Evaluation:**

* **TRIAL # 1** ~ Searching for word “**example**”
  + *Results:*

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Time (ms)** | **CPU Usage (ms)** | **Memory Usage (kb)** |
| ***Multithreading*** | 0 | 0 | 0 |
| ***Multiprocessing*** | 0 | 0 | 0 |

* + *Histogram:*

|  |  |
| --- | --- |
| **File** | **Occurrences** |
| bib.txt | 0 |
| paper1.txt | 0 |
| paper2.txt | 0 |
| progc.txt | 0 |
| progl.txt | 0 |
| progp.txt | 0 |
| trans.txt | 0 |

* + *Example:*
* **TRIAL # 2** ~ Searching for word “**program**”
  + *Results:*

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Time (ms)** | **CPU Usage (ms)** | **Memory Usage (kb)** |
| ***Multithreading*** | 0 | 0 | 0 |
| ***Multiprocessing*** | 0 | 0 | 0 |

* + *Histogram:*

|  |  |
| --- | --- |
| **File** | **Occurrences** |
| bib.txt | 0 |
| paper1.txt | 0 |
| paper2.txt | 0 |
| progc.txt | 0 |
| progl.txt | 0 |
| progp.txt | 0 |
| trans.txt | 0 |

* + *Example:*
* **TRIAL # 3** ~ Searching for word “**a**”
  + *Results:*

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Time (ms)** | **CPU Usage (ms)** | **Memory Usage (kb)** |
| ***Multithreading*** | 0 | 0 | 0 |
| ***Multiprocessing*** | 0 | 0 | 0 |

* + *Histogram:*

|  |  |
| --- | --- |
| **File** | **Occurrences** |
| bib.txt | 0 |
| paper1.txt | 0 |
| paper2.txt | 0 |
| progc.txt | 0 |
| progl.txt | 0 |
| progp.txt | 0 |
| trans.txt | 0 |

* + *Example:*
* Top 50 most frequent words (*according to developed system* *across all 7 files*):
  1. “**a**” ~ \_ occurrences
  2. “**the**” ~ \_ occurrences
  3. “**of**” ~ \_ occurrences
  4. “**and**” ~ \_ occurrences
  5. “**text**” ~ \_ occurrences
  6. “**end**” ~ \_ occurrences
  7. “**in**” ~ \_ occurrences
  8. “**is**” ~ \_ occurrences
  9. “**for**” ~ \_ occurrences
  10. “**to**” ~ \_ occurrences
  11. “**that**” ~ \_ occurrences
  12. “**be**” ~ \_ occurrences
  13. “**program**” ~ \_ occurrences
  14. “**this**” ~ \_ occurrences
  15. “**if**” ~ \_ occurrences
  16. “**I**” ~ \_ occurrences
  17. “**code**” ~ \_ occurrences
  18. “**text**” ~ \_ occurrences
  19. “**zone**” ~ \_ occurrences
  20. “**field**” ~ \_ occurrences
  21. “**window**” ~ \_ occurrences
  22. “**computer**” ~ \_ occurrences
  23. “**systems**” ~ \_ occurrences
  24. “**on**” ~ \_ occurrences
  25. “**bits**” ~ \_ occurrences
  26. “**code**” ~ \_ occurrences
  27. “**font**” ~ \_ occurrences
  28. “**int**” ~ \_ occurrences
  29. “**post**” ~ \_ occurrences
  30. “**look**” ~ \_ occurrences
  31. “**begin**” ~ \_ occurrences
  32. “**then**” ~ \_ occurrences
  33. “**hi**” ~ \_ occurrences
  34. “**var**” ~ \_ occurrences
  35. “**else**” ~ \_ occurrences
  36. “**edge**” ~ \_ occurrences
  37. “**programs**” ~ \_ occurrences
  38. “**system**” ~ \_ occurrences
  39. “**with**” ~ \_ occurrences
  40. “**false**” ~ \_ occurrences
  41. “**true**” ~ \_ occurrences
  42. “**boolean**” ~ \_ occurrences
  43. “**university**” ~ \_ occurrences
  44. “**systems**” ~ \_ occurrences
  45. “**report**” ~ \_ occurrences
  46. “**coding**” ~ \_ occurrences
  47. “**this**” ~ \_ occurrences
  48. “**which**” ~ \_ occurrences
  49. “**one**” ~ \_ occurrences
  50. “**code**” ~ \_ occurrences
* Average performance between multiprocessing and multithreading *(between 3 trials)*:

|  |  |  |  |
| --- | --- | --- | --- |
|  | **AVG Time (ms)** | **AVG CPU Usage (ms)** | **AVG Memory Usage (kb)** |
| ***Multithreading*** | 0 | 0 | 0 |
| ***Multiprocessing*** | 0 | 0 | 0 |

~ On average, you can see that \_ is slightly faster when it comes to the total time taken. However, the CPU usage averages in at \_, making \_ more efficient in that regard. Lastly, \_ tends to use somewhat less memory (measured in kilobytes), making it a better choice in that area.

**Final Discussion:**

The project demonstrates that **multithreading** generally performs better than **multiprocessing** for tasks involving I/O-bound operations like reading files and counting word occurrences, as threads share memory space and have less overhead compared to processes. However, multiprocessing may be more beneficial for CPU-bound tasks due to better CPU core utilization. One limitation of the current approach is the simplistic division of files into chunks, which may not align with line boundaries or word breaks, potentially leading to inaccurate word counts. To improve accuracy, a more sophisticated chunking method could be implemented, ensuring that chunks respect word boundaries. Additionally, better synchronization methods, like using condition variables instead of a simple mutex, could further optimize performance.

**APPENDIX:**

?????