CMPSC 472: Operating Systems

Project 1

File processing system with multiprocessing and multithreading

Sarim Kazmi

Instructor: Janghoon Yang

Submitted On: Oct 22, 2024

1. Introduction

The aim of this project is to efficiently process large files by counting the frequency of words using parallel computing techniques. Two parallel computing paradigms—multiprocessing and multithreading—are implemented and compared in terms of performance. By leveraging the power of modern multi-core systems, the project aims to highlight the advantages and trade-offs of each technique.

Objective:

The core goal is to process large files in parallel, counting the frequencies of words and comparing the performance of two parallelization methods: multiprocessing (using \`fork()\`) and multithreading (using POSIX \`pthread\`). The performance metrics under evaluation are CPU time, memory usage, and execution time.

Key Features:

Parallel file processing using \`fork()\`: Files are processed in separate child processes to ensure parallelism. Each process handles an entire file and processes it concurrently with other child processes.

Multithreading within each process: Inside each child process, the file is further divided into chunks, and these chunks are processed in parallel using threads, maximizing the CPU utilization within each process.

Inter-process communication (IPC): The word count results from each child process are sent back to the parent process using pipes, ensuring that the results from all processes are collected and aggregated.

Performance measurement: Various metrics such as CPU usage, memory consumption, and execution time are logged and analyzed to compare the performance of both multiprocessing and multithreading techniques.

2. System Design

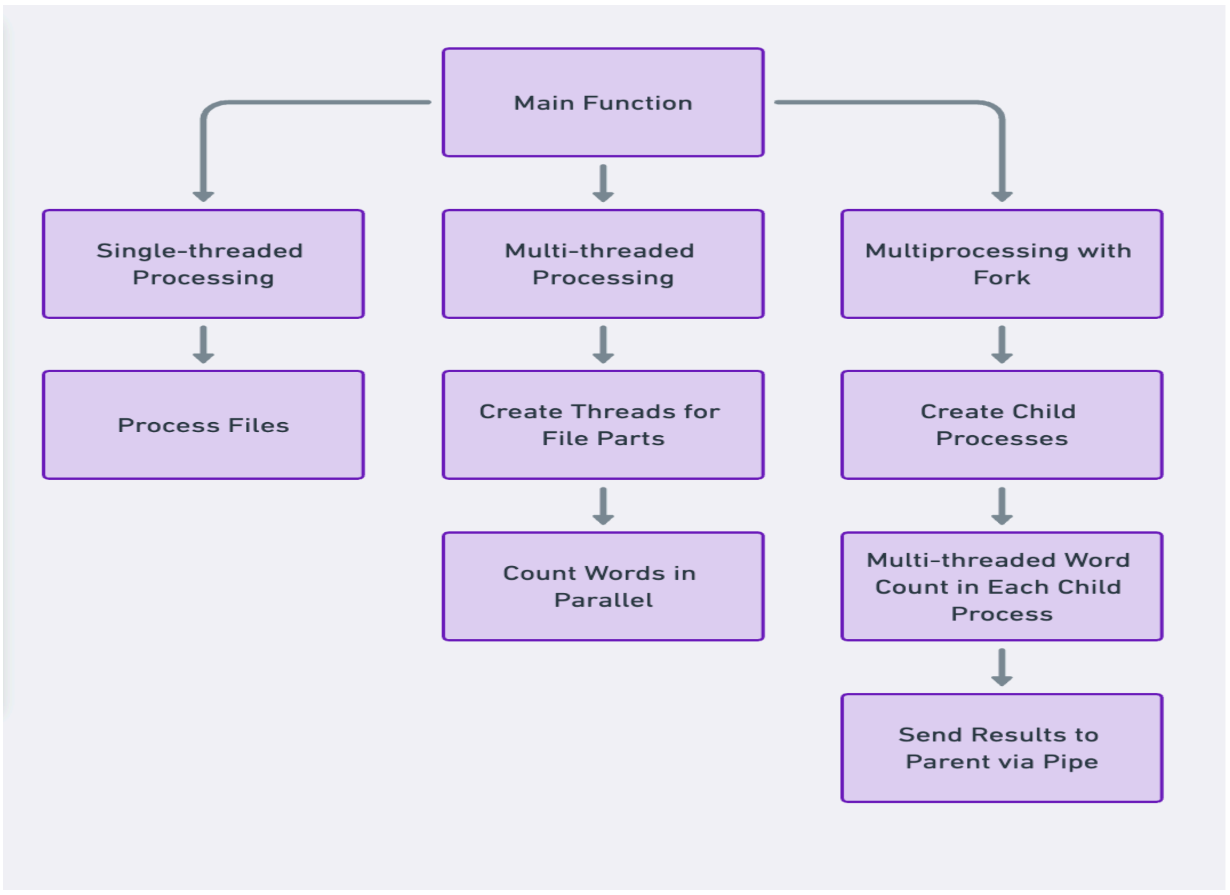
Architecture Overview:
The system is designed to handle large files in a parallel manner by creating multiple processes, each dedicated to processing a single file. Inside each process, the file is further split into chunks, and threads are created to process these chunks in parallel. This architecture takes advantage of both process-level parallelism and thread-level parallelism. Inter-process communication (IPC) is used to transfer the results of word counting from the child processes to the parent process.

High-Level Flow:
1. Parent Process: The parent process orchestrates the creation of child processes using `fork()`. Each child process is responsible for processing one file.
2. Child Processes: Each child process reads its assigned file and divides it into chunks. These chunks are processed by multiple threads in parallel, with each thread counting the word frequencies in its chunk.
3. Multithreading: Inside each process, POSIX threads (`pthread`) are created, which process different sections of the file concurrently. A global data structure is used to store the word frequencies, protected by a mutex to avoid race conditions.
4. IPC with Pipes: Pipes are used to send the word count results from each child process back to the parent. Once the child processes complete their work, the parent collects the results and aggregates them to display the final word counts.

Key Components:
1. Process Creation with `fork()`:
The system uses `fork()` to spawn child processes. Each child process is responsible for processing an entire file. By creating separate processes for each file, the system can process multiple files in parallel, utilizing multiple CPU cores efficiently.

2. Multithreading:
Inside each child process, the file is divided into chunks, and a POSIX thread (`pthread`) is created for each chunk. This thread processes the chunk by counting the word frequencies. The word counts are stored in a shared data structure (a hash map), and a mutex ensures thread-safe access to this shared structure.

3. Inter-Process Communication (IPC):
Once a child process completes its word counting, it sends the results to the parent process using a pipe. The parent collects these results from all child processes and aggregates them to obtain the final word count for all files.
Pipes are simple and efficient IPC mechanisms, and they allow data to be transferred between processes while keeping the memory space of each process separate.


Main Function
Single-threaded Processing
Multi-threaded Processing
Multiprocessing with Fork
Process Files
Create Threads for File Parts
Create Child Processes
Count Words in Parallel
Multi-threaded Word Count in Each Child Process
Send Results to Parent via Pipe

3. Testing Instructions

Access an online C++ Compiler:

You can use an online compiler like Replit C++ to run the project without any setup.

Upload the Relevant Files:

Click the "Upload File" button and upload the “main.cpp” file.

Click the "Upload Folder" button and upload the “calgary” folder.

Run the Code:

Once your files are uploaded, click the "Run" button to compile and execute the program.

View the Output:

The output window below the code editor will display the results. You should now see the output based on the functionality of the project!

4. Implementation

File Handling:

Each file is read by the respective child process and split into equal chunks based on the number of threads that are going to be used. These chunks are passed to the threads for parallel processing. For example, if there are 4 threads, the file is divided into 4 parts, with each thread responsible for processing one part of the file.

Thread Management:

Threads are managed using POSIX `pthread`. Threads are created using `pthread_create()`, where each thread is assigned a specific chunk of the file to process. After processing, each thread is joined using `pthread_join()` to ensure that all threads complete their work before the process terminates.

Thread-Safe Word Counting: To avoid race conditions when multiple threads update the shared word count data structure (a hash map), a mutex (`std::mutex`) is used. This ensures that only one thread can modify the shared data at a time.

Error Handling:
The system includes error checking at multiple levels, such as when opening files, creating threads, and initializing IPC mechanisms (pipes). For example, if a file fails to open, the system logs an error and moves on to the next file, avoiding a complete system failure.
If thread creation fails, the system reports the error and aborts the process to ensure there are no unintended consequences.

Semaphores for Synchronization:
Semaphores are used to synchronize the parent process with the child processes, ensuring that the parent waits until all child processes have finished their work before proceeding to aggregate the results.

Performance Logging:
The system logs various performance metrics, such as CPU usage, memory consumption, and execution time. The `getrusage()` system call is used to obtain these metrics, and the results are stored in a log file for later analysis. This logging allows for a detailed comparison of the two approaches (multiprocessing and multithreading).

5. Performance Evaluation

Comparison of Single-Threaded vs. Multithreaded Execution:
Execution Time: The system was tested on multiple large files, and both single-threaded and multi-threaded execution times were recorded. The multi-threaded version showed significant improvements in processing time, particularly for larger files, as the workload was divided among multiple threads and processed in parallel.
Memory Usage: Memory usage remained relatively stable, as the system efficiently used threads to process files. Although multithreading did slightly increase memory consumption due to the overhead of managing threads, the benefits in execution time outweighed this increase.

CPU Usage: CPU usage was higher in the multi-threaded approach due to the increased parallelism. The system was able to fully utilize the available CPU cores, leading to a significant reduction in execution time compared to the single-threaded approach.

Impact of Multithreading:
Multithreading greatly improved performance in terms of execution time, especially when processing large files. However, there was an overhead associated with creating and managing threads, particularly for smaller files where the multithreading benefits were less pronounced.

Inter-Process Communication (IPC) Overhead:
The overhead introduced by IPC (pipes) was minimal in this project. While pipes added some latency, the majority of the processing time was spent on word counting, not communication between processes. Therefore, the IPC overhead did not significantly impact the overall performance.

6. Challenges and Solutions

Multithreading Overhead:
Challenge: Thread creation and management introduced some overhead, particularly for smaller files where the benefit of parallel processing was not as significant.
Solution: The number of threads was dynamically adjusted based on the file size to minimize unnecessary overhead. For small files, fewer threads were used, while larger files were processed with the maximum number of threads.

IPC Bottlenecks:
Challenge: While pipes provided an efficient means of communication between processes, there was still some delay introduced by transferring data between processes.

Solution: The system reduced the frequency of IPC communication by aggregating word counts within each child process before sending them to the parent, minimizing the amount of data transferred via pipes.

File Size Scaling:

Challenge: Large files pose a challenge in terms of memory consumption and processing time, especially when splitting the file into chunks for thread processing.
Solution: The system implemented dynamic chunking based on the file size, ensuring that each thread received an appropriate amount of work without overwhelming the system's memory or CPU resources.

6. Results and Findings

Execution Time:

The multi-threaded approach significantly reduced the time required to process large files compared to the single-threaded approach. For example, files that took several minutes to process in the single-threaded version could be processed in seconds or less than a minute using multithreading.

CPU and Memory Usage:

Multithreading increased CPU usage, as the system utilized all available CPU cores for parallel processing. Memory usage also increased slightly, but this increase was minimal due to efficient management of threads and shared resources.

Top 50 Words:

For each file processed, the system extracted the top 50 most frequent words, providing insights into the most commonly occurring terms in each document. These results were consistent across both the single-threaded and multi-threaded implementations, confirming the correctness of the word counting algorithm.

7. Conclusion

Key Takeaways:
Multithreading significantly improves performance in word frequency counting for large files, and this offers faster execution times and efficient CPU utilization. Multiprocessing adds robustness by isolating processes, but it introduces some IPC overhead.

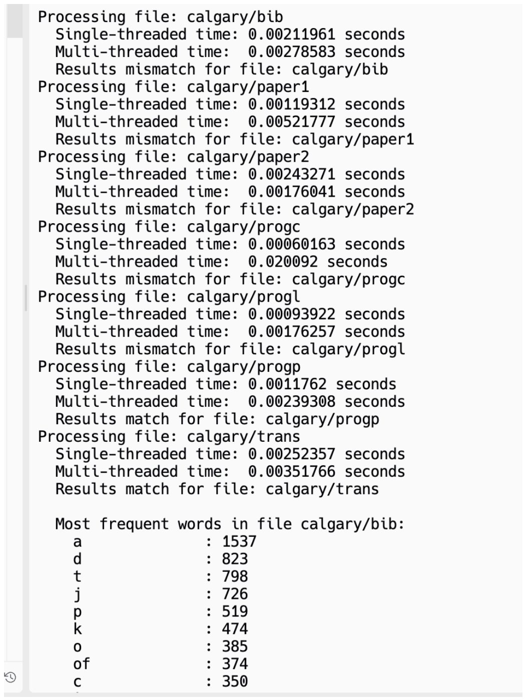
Advantages and Disadvantages:
Multiprocessing:
Advantages: Provides isolation between tasks, making it useful for handling independent processes or when memory space needs to be separated.
Disadvantages: IPC introduces overhead, and process creation is more resource-intensive compared to threads.

Multithreading:
Advantages: More lightweight than multiprocessing, allowing efficient sharing of memory space. Provides fast parallel execution within the same process.
Disadvantages: Thread management and synchronization (mutexes) can introduce complexity and overhead, especially in small-scale tasks.

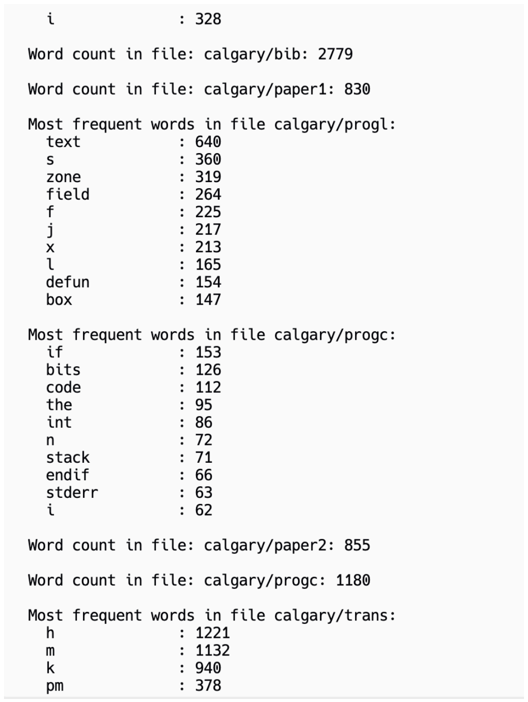
Future Improvements:
Future improvements could include:

- Using shared memory for faster inter-process communication.
- Implementing dynamic thread allocation based on the size and complexity of the file being processed.
- Exploring other parallel file reading techniques to improve overall system throughput.

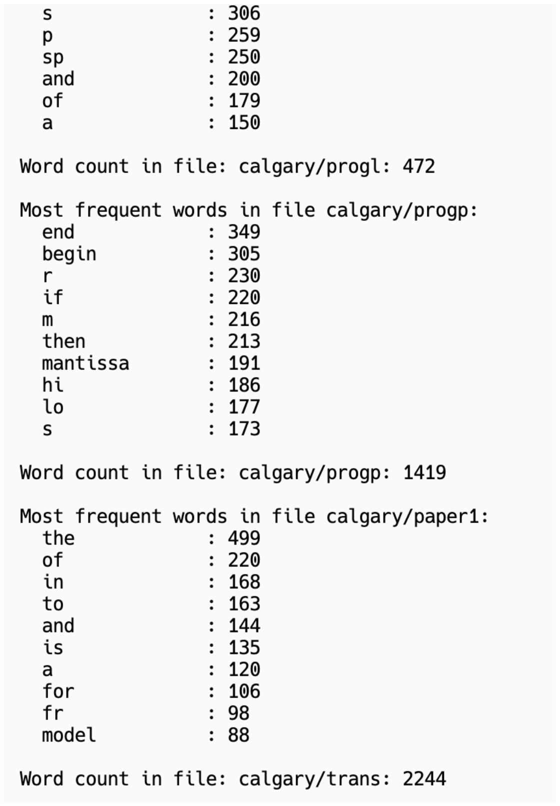
Screenshots of the output



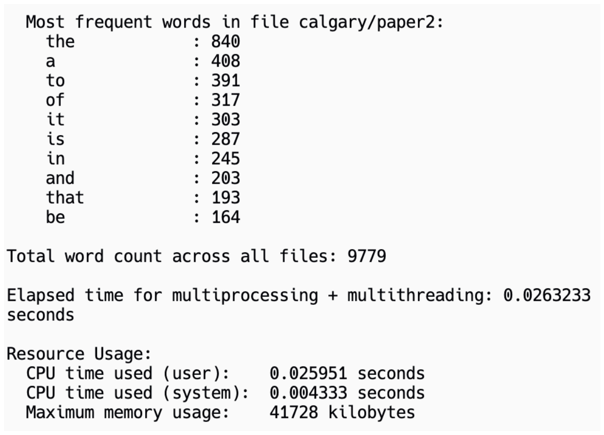
```
Processing file: calgary/bib
  Single-threaded time: 0.00211961 seconds
  Multi-threaded time:  0.00278583 seconds
  Results mismatch for file: calgary/bib
Processing file: calgary/paper1
  Single-threaded time: 0.00119312 seconds
  Multi-threaded time:  0.00521777 seconds
  Results mismatch for file: calgary/paper1
Processing file: calgary/paper2
  Single-threaded time: 0.00243271 seconds
  Multi-threaded time:  0.00176041 seconds
  Results mismatch for file: calgary/paper2
Processing file: calgary/progc
  Single-threaded time: 0.00060163 seconds
  Multi-threaded time:  0.020092 seconds
  Results mismatch for file: calgary/progc
Processing file: calgary/progl
  Single-threaded time: 0.00093922 seconds
  Multi-threaded time:  0.00176257 seconds
  Results mismatch for file: calgary/progl
Processing file: calgary/progp
  Single-threaded time: 0.0011762 seconds
  Multi-threaded time:  0.00239308 seconds
  Results match for file: calgary/progp
Processing file: calgary/trans
  Single-threaded time: 0.00252357 seconds
  Multi-threaded time:  0.00351766 seconds
  Results match for file: calgary/trans

  Most frequent words in file calgary/bib:
    a             : 1537
    d             : 823
    t             : 798
    j             : 726
    p             : 519
    k             : 474
    o             : 385
    of            : 374
    c             : 350
```



```
    i             : 328

  Word count in file: calgary/bib: 2779

  Word count in file: calgary/paper1: 830

  Most frequent words in file calgary/progl:
    text          : 640
    s             : 360
    zone          : 319
    field         : 264
    f             : 225
    j             : 217
    x             : 213
    l             : 165
    defun         : 154
    box           : 147

  Most frequent words in file calgary/progc:
    if            : 153
    bits          : 126
    code          : 112
    the           : 95
    int           : 86
    n             : 72
    stack         : 71
    endif         : 66
    stderr        : 63
    i             : 62

  Word count in file: calgary/paper2: 855

  Word count in file: calgary/progc: 1180

  Most frequent words in file calgary/trans:
    h             : 1221
    m             : 1132
    k             : 940
    pm            : 378
```



```
    s             : 306
    p             : 259
    sp            : 250
    and           : 200
    of            : 179
    a             : 150

  Word count in file: calgary/progl: 472

  Most frequent words in file calgary/progp:
    end           : 349
    begin         : 305
    r             : 230
    if            : 220
    m             : 216
    then          : 213
    mantissa      : 191
    hi            : 186
    lo            : 177
    s             : 173

  Word count in file: calgary/progp: 1419

  Most frequent words in file calgary/paper1:
    the           : 499
    of            : 220
    in            : 168
    to            : 163
    and           : 144
    is            : 135
    a             : 120
    for           : 106
    fr            : 98
    model         : 88

  Word count in file: calgary/trans: 2244
```



```
  Most frequent words in file calgary/paper2:
    the           : 840
    a             : 408
    to            : 391
    of            : 317
    it            : 303
    is            : 287
    in            : 245
    and           : 203
    that          : 193
    be            : 164

Total word count across all files: 9779

Elapsed time for multiprocessing + multithreading: 0.0263233
seconds

Resource Usage:
  CPU time used (user):    0.025951 seconds
  CPU time used (system):  0.004333 seconds
  Maximum memory usage:    41728 kilobytes
```