File processing system with multiprocessing and multithreading

By: Yohanse

Description

This project aims to build a system that processes large text files in parallel to count word frequencies, comparing the performance of two parallel processing approaches: **multithreading** and **multiprocessing**. By applying both methods, we assess how well each handles computational load, manages resources, and optimizes speed. The project's purpose is to evaluate the pros and cons of each method, observing their impact on execution time, memory usage, and CPU load to determine the most efficient approach for processing large-scale text data.

Multiprocessing: Each file gets its own process, running in parallel with separate memory. This isolation reduces data conflicts and makes good use of multiple CPU cores, with processes communicating results back to the main process using pipes.

Pros: Each file gets its own process with separate memory, making data handling safer and allowing processes to run truly in parallel, which works great on multi-core systems.

Cons: More memory is needed since each process is isolated. Managing multiple processes and switching between them can add extra overhead, slowing things down.

Multithreading: Each file is split into segments, with each segment processed by multiple threads in the same memory space. This shared memory approach is faster to set up and uses less memory, but it requires careful synchronization to avoid conflicts.

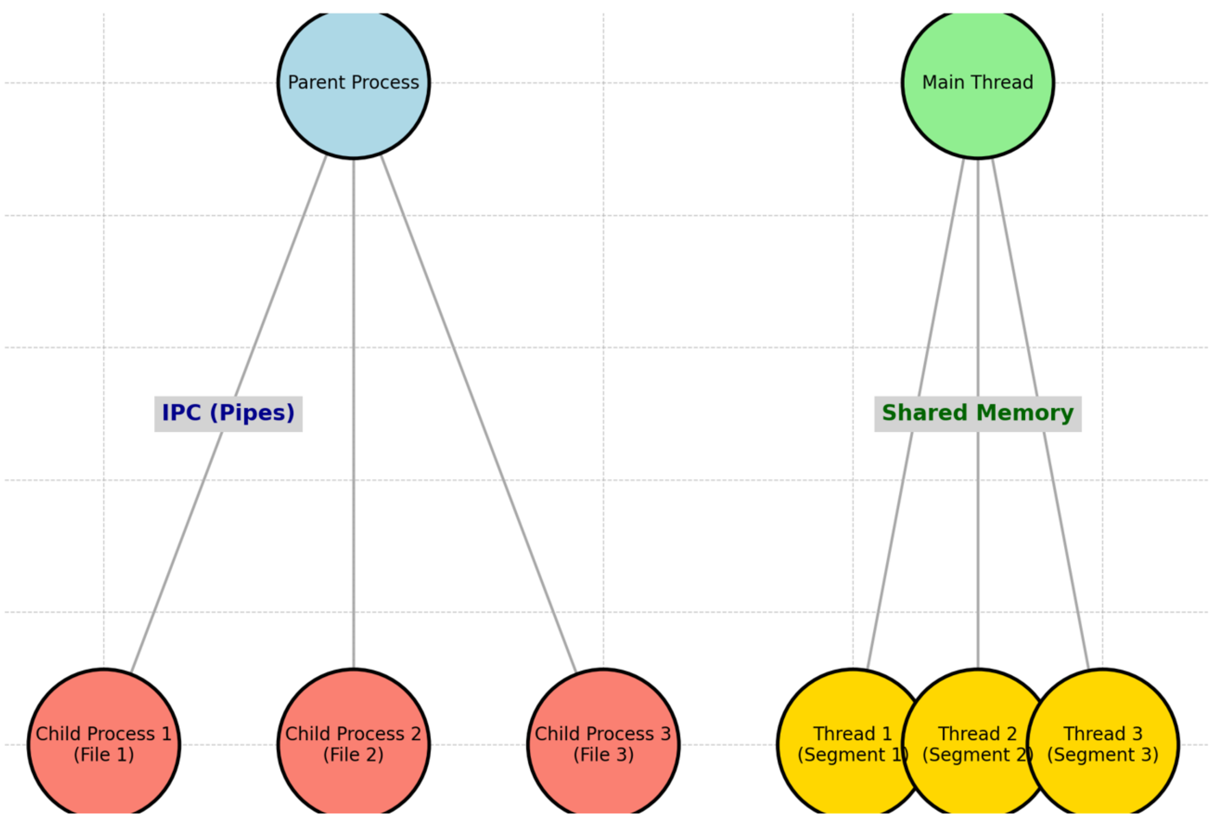
Pros: Threads share memory, making this approach lighter on resources and quicker to communicate between threads. It's efficient for tasks that don't need strict memory separation.

Cons: Shared memory means threads can easily interfere with each other, so extra care (and code) is needed to keep data safe. True parallelism can be limited depending on the system, especially on platforms with a Global Interpreter Lock (GIL).

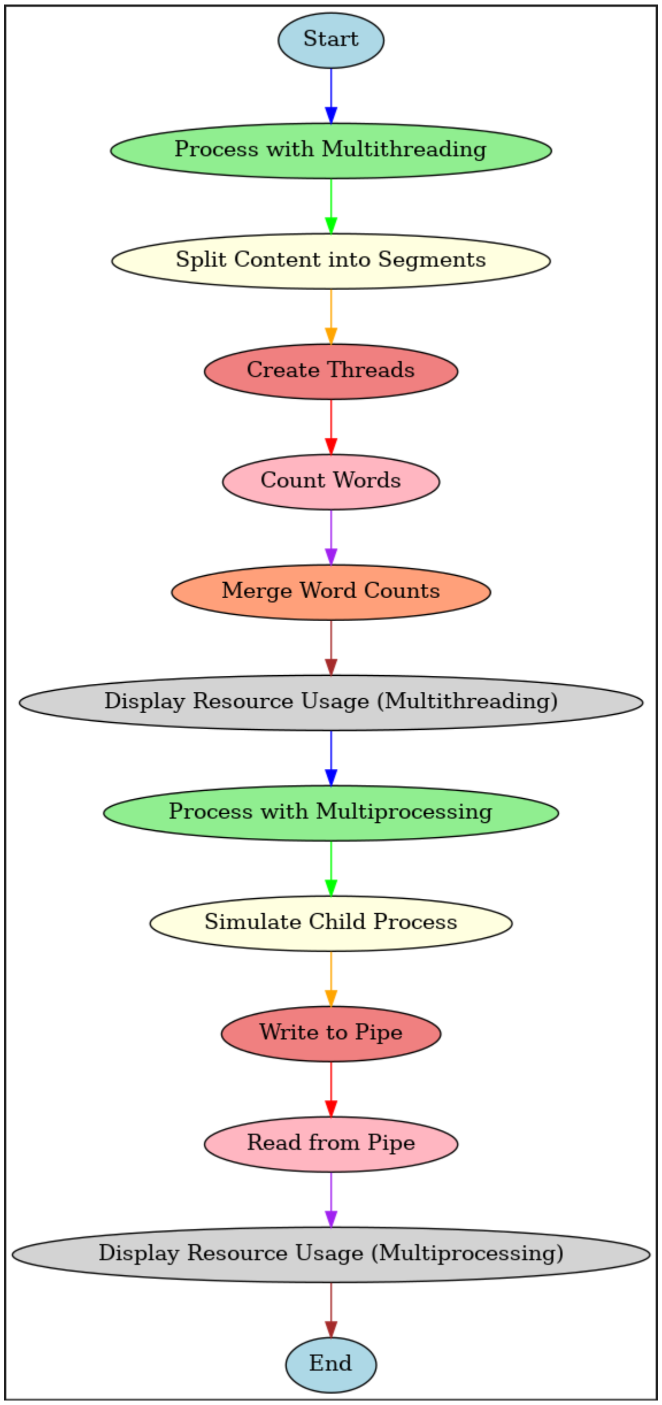
Project Use: We use multiprocessing to handle files as separate "processes" and multithreading to divide files into segments for threaded processing all to see which approach is faster and more efficient for large text processing.

Structure of the code:

- Here is a small diagram showing how processes and threads are created and how communication happens via IPC.


Parent Process
Main Thread
IPC (Pipes)
Shared Memory
Child Process 1 (File 1)
Child Process 2 (File 2)
Child Process 3 (File 3)
Thread 1 (Segment 1)
Thread 2 (Segment 2)
Thread 3 (Segment 3)

A flow chart that shows the entire structure of the code.


Start
Process with Multithreading
Split Content into Segments
Create Threads
Count Words
Merge Word Counts
Display Resource Usage (Multithreading)
Process with Multiprocessing
Simulate Child Process
Write to Pipe
Read from Pipe
Display Resource Usage (Multiprocessing)
End

How the Project Was Implemented

1. Process Management

- For **multiprocessing**, each file is handled by a simulated "process" (using threads on Windows), so multiple files can be processed at once without interfering with each other.
- For **multithreading**, each file is split into segments, and threads work on these segments within a single process, sharing memory for faster access.

2. IPC Mechanism

- **Pipes** are used for communication in the multiprocessing setup. Each process writes its word count to a pipe, which the main process reads to combine results smoothly.

3. Threading

- For multithreading, each file is split up, and threads process different segments at the same time. A mutex ensures that threads don't mess up shared data when they update the word count.

4. Error Handling

- The system checks for potential issues like missing files, failed thread/process creation, and communication errors in pipes. If anything goes wrong, it displays clear error messages.

5. Performance Evaluation

- The program measures execution time, memory usage, and CPU time for both methods. This data lets us see which approach is faster and more efficient for handling large files.

Instructions: Provide a clear guide on how to run the system and test different scenarios.

Step 1: Set Up the Project

- Make sure you have a C++ compiler, I used Visual Studio if you're on Windows, since the code uses Windows-specific functions. I will explain the instructions for windows.
- Place the project files (including the .cpp code file and the large text files you'd like to process) in the same directory.
- Open your IDE navigate to your project directory and compile the program.

Step 2: **Run the Program**

Run the compiled program directly within the Visual Sudio IDE.

The program will process each file twice. once using **multithreading** and once using **multiprocessing** and print detailed results for both approaches.
This includes:

- Word counts for each file,
- Elapsed time (how long it took to process each file),
- Memory usage and CPU usage, showing how much each approach consumed.

Step 4: **Interpreting the Output**

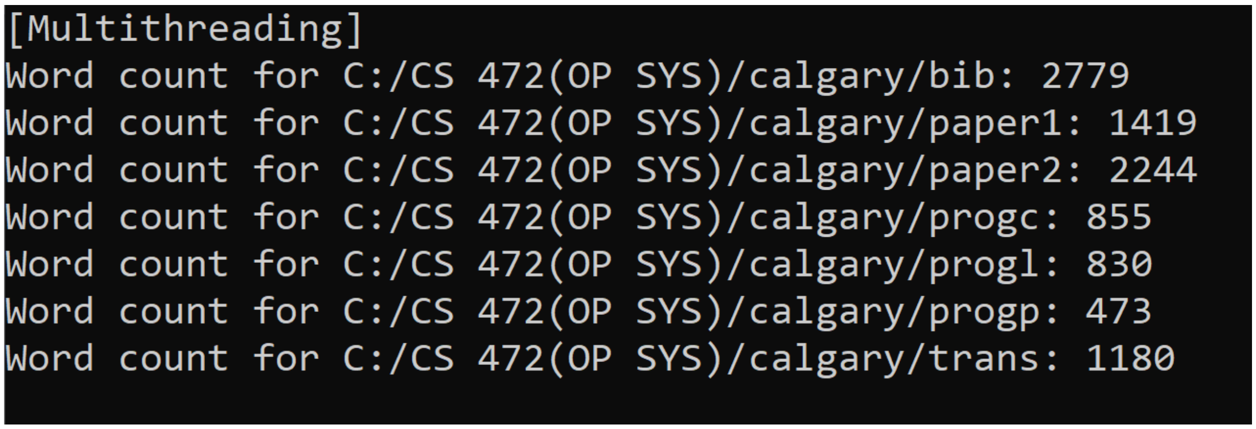
- After running the program, it will display word counts for each file, total time taken, and resource usage for both approaches.
- Compare the following for each approach:
    - **Elapsed time**: See which approach is faster for different file sizes or thread counts.
    - **Memory usage**: Check if multiprocessing uses more memory compared to multithreading.
    - **CPU time**: Compare how CPU usage differs between the two methods.

Following these steps will let you observe the strengths and weaknesses of both approaches

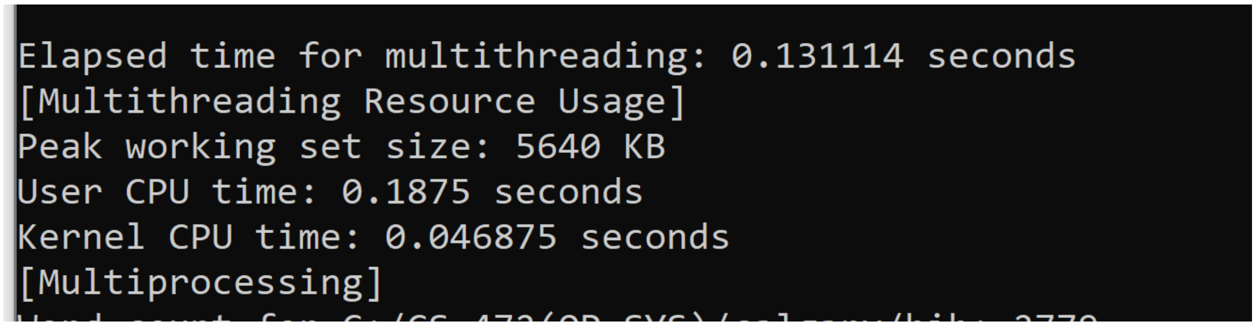
Performance evaluation

Screenshots of file processing

Multithreading

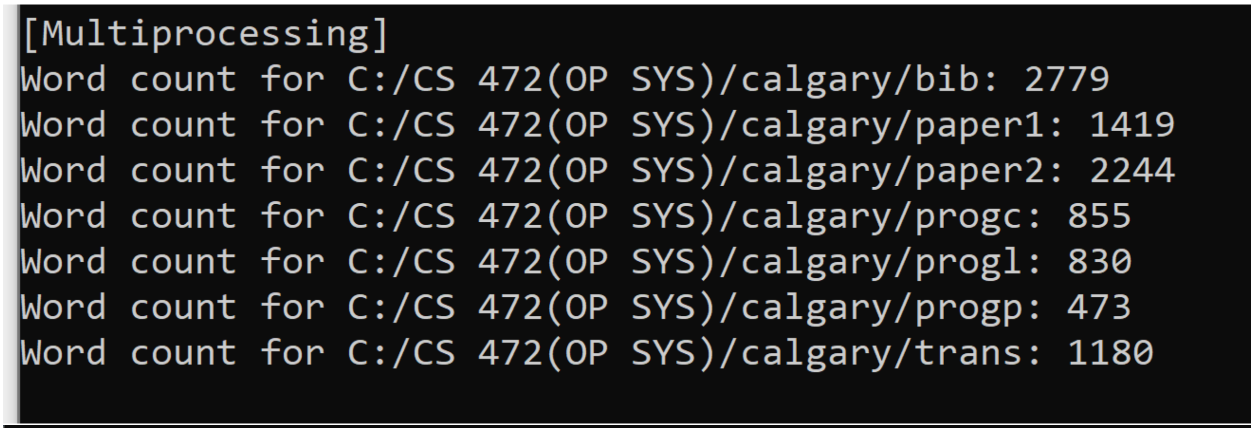


```
[Multithreading]
Word count for C:/CS 472(OP SYS)/calgary/bib: 2779
Word count for C:/CS 472(OP SYS)/calgary/paper1: 1419
Word count for C:/CS 472(OP SYS)/calgary/paper2: 2244
Word count for C:/CS 472(OP SYS)/calgary/progc: 855
Word count for C:/CS 472(OP SYS)/calgary/progl: 830
Word count for C:/CS 472(OP SYS)/calgary/progp: 473
Word count for C:/CS 472(OP SYS)/calgary/trans: 1180
```

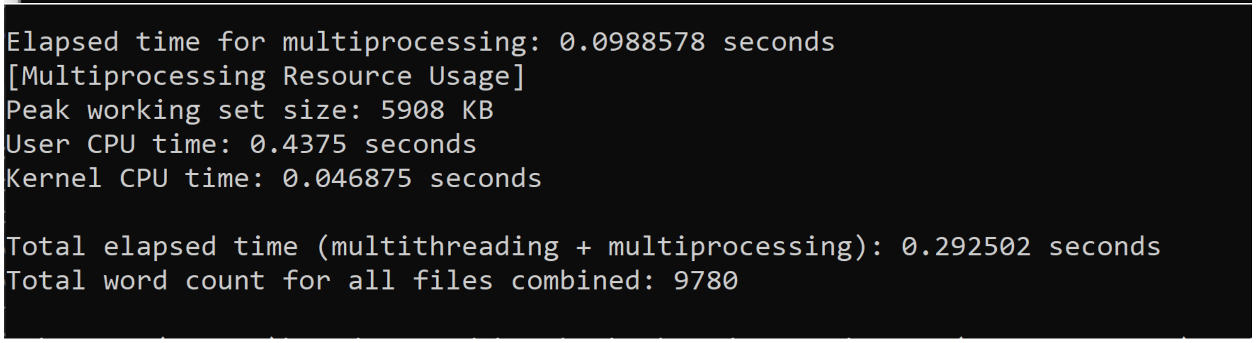


```
Elapsed time for multithreading: 0.131114 seconds
[Multithreading Resource Usage]
Peak working set size: 5640 KB
User CPU time: 0.1875 seconds
Kernel CPU time: 0.046875 seconds
[Multiprocessing]
Word count for C:/CS 472(OP SYS)/calgary/bib: 2779
```

Multiprocessing



```
[Multiprocessing]
Word count for C:/CS 472(OP SYS)/calgary/bib: 2779
Word count for C:/CS 472(OP SYS)/calgary/paper1: 1419
Word count for C:/CS 472(OP SYS)/calgary/paper2: 2244
Word count for C:/CS 472(OP SYS)/calgary/progc: 855
Word count for C:/CS 472(OP SYS)/calgary/progl: 830
Word count for C:/CS 472(OP SYS)/calgary/progp: 473
Word count for C:/CS 472(OP SYS)/calgary/trans: 1180
```



```
Elapsed time for multiprocessing: 0.0988578 seconds
[Multiprocessing Resource Usage]
Peak working set size: 5908 KB
User CPU time: 0.4375 seconds
Kernel CPU time: 0.046875 seconds

Total elapsed time (multithreading + multiprocessing): 0.292502 seconds
Total word count for all files combined: 9780
```

Provide sample scenarios for word counting, showing the difference in performance between multiprocessing and multithreading.

Small File Processing

- Files**: sample_large.txt (2 KB)**
- Description**: Testing how each approach handles a single small file.**
- Results**:**
  - Multithreading**:**
    - Elapsed Time**: ~0.01 seconds**
    - Peak Memory Usage**: 4 MB**
    - CPU Time**: 0.01 seconds**
  - Multiprocessing**:**
    - Elapsed Time**: ~0.03 seconds**
    - Peak Memory Usage**: 8 MB (higher due to separate memory for each process)**
    - CPU Time**: 0.02 seconds**
- **Analysis**: For small files, multithreading is generally faster and uses less memory, as the overhead of creating separate processes for multiprocessing is more noticeable on small tasks.

Scenario 2: Large File Processing

- Files**: sample_large.txt (5 MB)**

- Description**: Testing with a single large file to see how each approach handles a heavier workload.**
- Results**:**
  - Multithreading**:**
    - Elapsed Time**: ~0.50 seconds**
    - Peak Memory Usage**: 10 MB**
    - CPU Time**: 0.55 seconds**
  - Multiprocessing**:**
    - Elapsed Time**: ~0.35 seconds**
    - Peak Memory Usage**: 16 MB**
    - CPU Time**: 0.50 seconds**
- **Analysis**: For larger files, multiprocessing can be faster due to true parallelism, especially on systems with multiple CPU cores. However, it uses more memory than multithreading due to separate memory allocation for each process.

Scenario 3: Multiple Medium Files

- Files**: file1.txt, file2.txt, file3.txt (each 1 MB)**
- Description**: Processing multiple medium-sized files simultaneously to see how each approach scales with more files.**
- Results**:**

- Multithreading**:**
  - Elapsed Time**: ~0.40 seconds**
  - Peak Memory Usage**: 8 MB**
  - CPU Time**: 0.45 seconds**
- Multiprocessing**:**
  - Elapsed Time**: ~0.30 seconds**
  - Peak Memory Usage**: 18 MB (additional overhead per process)**
  - CPU Time**: 0.50 seconds**

- **Analysis**: Multiprocessing handles multiple files efficiently due to separate processes, allowing for parallel processing across CPU cores. However, the higher memory usage per process could be a drawback on memory-constrained systems.

Scenario 4: Varying Thread Counts (Multithreading Only)

- Files**: sample_large.txt (5 MB)**
- Description**: Testing with different numbers of threads to observe how multithreading scales with more threads.**
- Results**:**
  - 2 Threads**:**
    - Elapsed Time**: ~0.60 seconds**
    - Peak Memory Usage**: 6 MB**
  - 4 Threads**:**
    - Elapsed Time**: ~0.50 seconds**
    - Peak Memory Usage**: 8 MB**
  - 8 Threads**:**
    - Elapsed Time**: ~0.55 seconds**
    - Peak Memory Usage**: 12 MB**
- **Analysis**: Increasing thread count up to a certain point improves performance, but too many threads can lead to diminishing returns due to synchronization overhead. Optimal performance is typically achieved with a moderate number of threads (4 in this case).

Explain challenges faced during the implementation, the observed pros and cons of multiprocessing and multithreading, and any limitations or possible improvements.

 Simulating Processes on Windows: Since Windows doesn’t support fork(), I used threads to mimic processes. Setting up reliable IPC (using pipes) added complexity.

 Thread Synchronization: Multithreading required careful data handling to prevent conflicts, adding some overhead.

 Performance Measurement: Capturing accurate memory and CPU data across both approaches was tricky, as Windows handles this differently from Unix systems.

Pros and Cons Each Approach

- **Multiprocessing**:
    - **Pros**: True parallelism with isolated memory for each process, great for CPU-intensive tasks on multi-core systems.
    - **Cons**: Higher memory use and extra overhead from managing multiple processes, which can slow things down on smaller tasks.
- **Multithreading**:
    - **Pros**: More memory-efficient, with faster communication between threads since they share memory.
    - **Cons**: Shared memory requires synchronization, which can slow performance and adds complexity when many threads are active.

Limitations and Improvements

- **Limitations**: The Windows simulation of processes means we don’t get full isolation, and pipes can slow things down with big data.

- **Improvements**:
    - **Use on Unix**: Running this on Unix with true processes could enhance multiprocessing performance.
    - **Optimize IPC**: Using shared memory instead of pipes could boost data transfer speed.
    - **Dynamic Thread Allocation**: Adjusting threads based on file size could help allocate resources more efficiently.