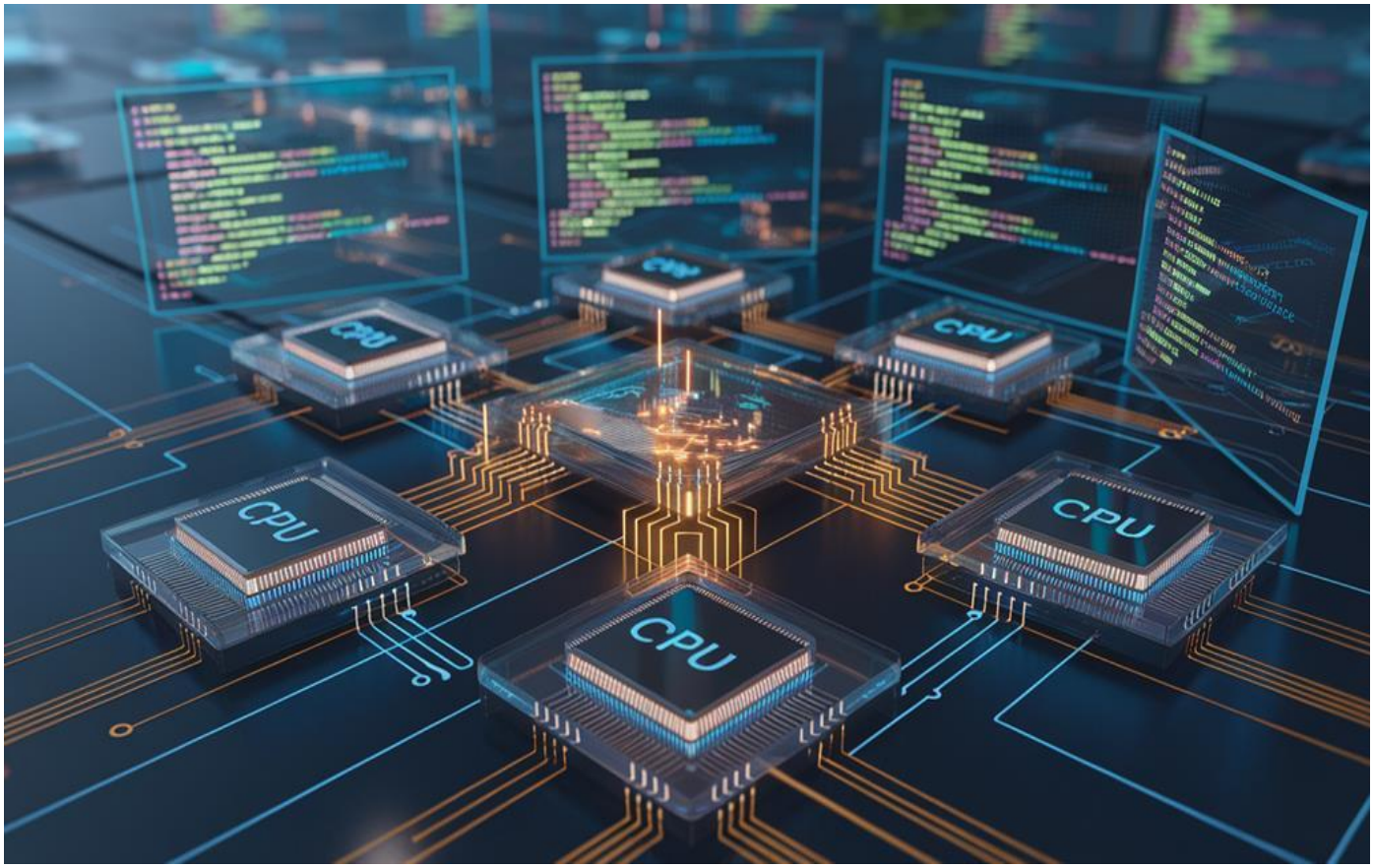
WORD STATISTICS



Operating System-2

Contents:

1. Project Description
2. Problem Modeling
3. GUI Design
4. Implementation Details

Project Description

- This project is a multi-threaded program designed to read text files from a specific directory and generate a real-time word-statistics

- The program allows users to choose a directory, with an option to include or exclude subdirectories

- Once a directory is selected, the program automatically detects all text files (txt format) located inside and display them on the GUI

- The core purpose of the program is to create a detailed word statistics for each file and the entire directory

- These statistics include:
    - Total number of words per file
    - Number of “is”, “are” and “you”
    - Longest and shortest word within each file
    - Longest and shortest word across whole directory

- To improve performance and responsiveness, the program uses multi-threading

- The main thread handles directory exploration and file discovery

- The additional worker threads (based on number of available CPU cores) process the text files in parallel

- This ensures faster processing, especially when dealing with large number of files

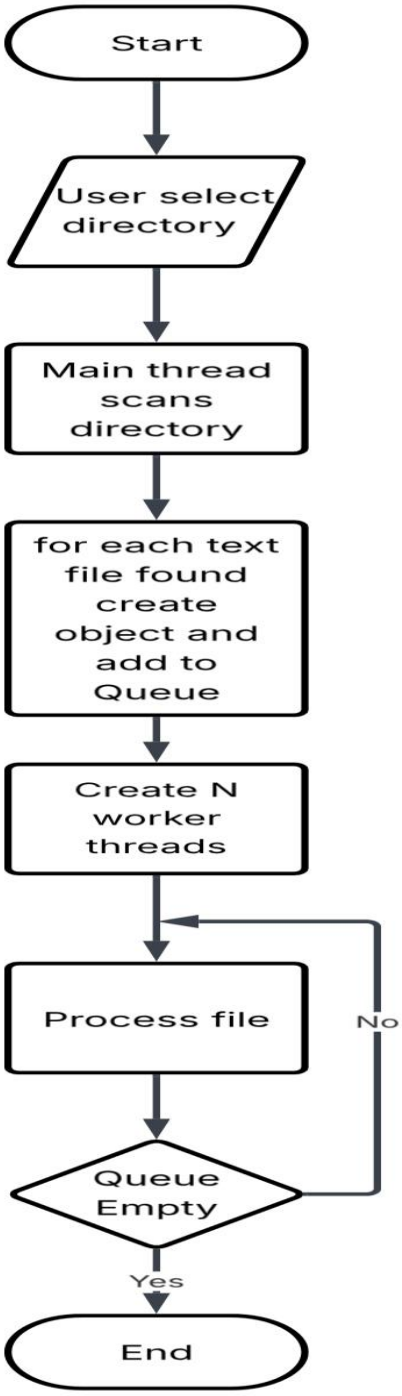
- As each thread analyzes it’s assigned files, it sends continuous updates back to the GUI

- The statistics are displayed in real-time, allowing user to monitor progress of each file as it is being processed

- The GUI presents the results in a table, making it easier to compare and visualize data

Overall, this project demonstrates efficient directory traversal, file processing, GUI, and multi-threaded (parallel) programming. It provides a practical tool for examining text files while highlighting the benefits of parallelism

Problem Modeling



The problem involves reading all text files inside a selected directory and computing word statistics in real-time while using multi-threading. To solve this efficiently, the system must separate tasks of directory scanning, file processing, and GUI updates.

Next, we will describe the logical structure and behavior of the systems

1. Main Thread Responsibilities:

The main thread is responsible for preparing the environment and managing the initial workflow

- Accept the directory path from GUI
- Determines whether subdirectory search is enabled *isSelected()*
- Scans the selected directory and subdirectories if checked *directoryExplorer.explore(dipath, recursive)*
- Identifies all txt files *isFile()* and create object *FileResult* for each file
- Pushes created objects into a shared *ConcurrentLinkedQueue fileResultsQueue*

- Display each discovered file name in the GUI in real-time *updateCell()*

- Start worker threads that will process these files *exploreFiles()*

This thread doesn't process the content of the files itself, it acts mainly as the factory

2. Worker Threads (File Processing Threads):

The system uses multi-threading to process multiple files concurrently. The number of threads is based on the number of available CPU cores to maximize performance

- Continuously polls the shared queue for object *FileResult*

- Open and read the text file line by line *readline()*

- Break down the text into words *words=line.split(" ")*

- Updates:
    - Total number of words
      incrementNumOfWords()
    - Count of “is”, “are”, “you”
      incrementNumOfIsWords()
      incrementNumOfAreWords()
      incrementNumOfYouWords()
    - Longest and shortest word per file
      compareAndSetIfLonger()
      compareAndSetIfShorter()

- Updates shared:
    - Longest word and Shortest
      updateAndGet() using thread-safe
      “*AtomicRefrence*” variables

- Send updates back to the GUI so user can see progress in real-time
  updateFileResult(fileResult, line)

- Stops immediately if main thread set “*stopped*” to true

This model ensures parallelism while preventing GUI freezing

3. Shared Data Structure:

To coordinate work between threads, the following is used:

- *ConcurrentLinkedQueue<FileResult>* Shared queue containing files waiting to be processed, worker threads pull from it without blocking main
- *AtomicRefrence<String> Longest/Shortest* Stores the longest and shortest word found in directories
- *AtomicBoolean stopped* Indicates whether user stopped process or not (when closing program)

All shared structures are thread-safe to avoid race conditions

4. GUI Interaction Model:

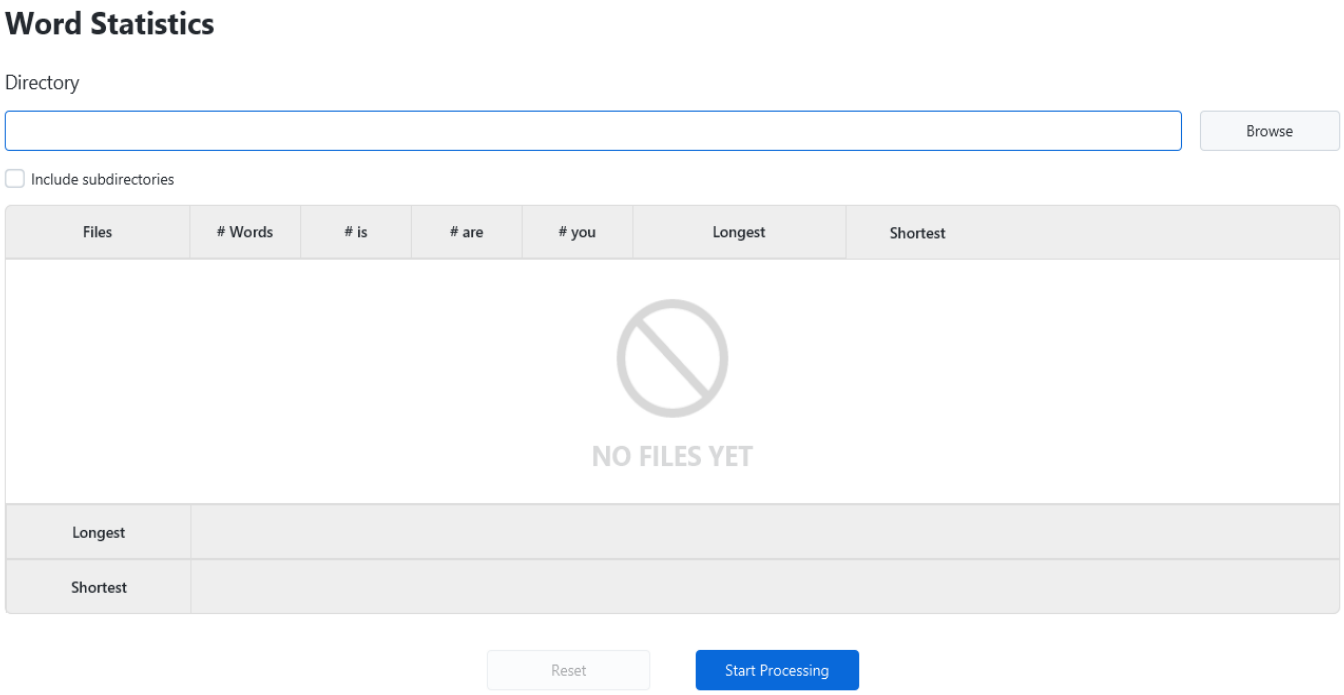
GUI is updated in real-time as threads process files

- When main thread finds a new file, it's displayed immediately
- When a worker thread updates a statistic the table refresh
- When a thread changes the global longest or shortest word the statistics of directory updates

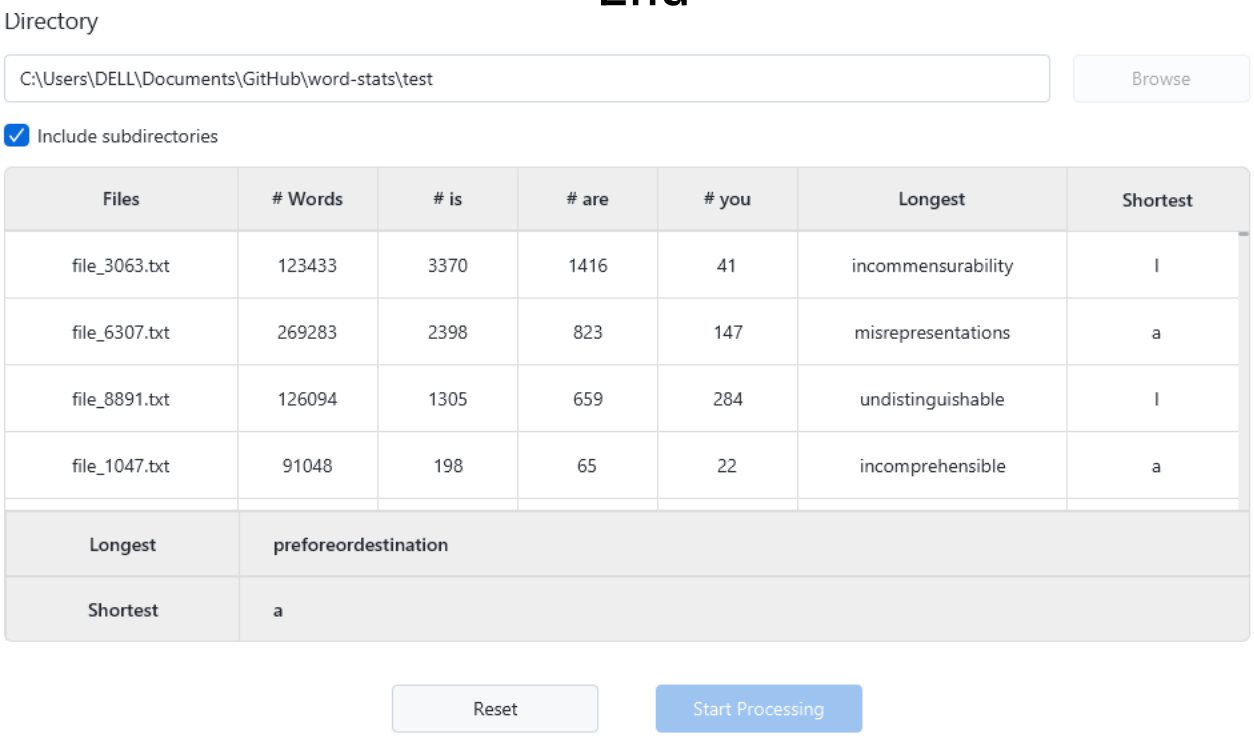
This ensures the user sees the program's progress dynamically without waiting for the files to finish

GUI Design

Start



End



Implementation

How the system was developed, the logic behind key components, and how multithreading, file processing, and GUI updates were implemented

1. Directory scanning logic:

The main thread begins by reading directory given by user then:

- Validate path `isDirectoryExists()`
- Check whether subdirectories are enabled `subDirectoriesCheckBox.isSelected()`
- Scan directory for txt files `file.isFile()&&file.getName().endsWith(".txt")`
    - If subdirectories selected a recursive is performed `if(recursive&&file.isDirectory())`

```
{exploreDirectoryRecursively(file.getAbso
lutePath(), true);}
```

- For each txt file found, A FileResult object is created and pushed into *ConncurrentLinkedQueue* and updated to GUI

```
FileResult fileResult=new
FileResult(file.getAbsolutePath());
fileResultsQueue.add(fileResult);
fileResults.add(fileResult);
```

2. Multi-threading:

To speed up processing:

- Thread Count:
  - Number of worker threads is equal to number of CPU cores `Runtime.getRuntime().availableProcessors()`

- Each thread executes a *FileExplorerRunnable* which:
  - Continuously retrieve items from shared queue `fileResult = fileResultsQueue.poll()`
  - Process file assigned to it

- Updates global and local statistics
- Trigger GUI updates
- Stops if stopped is set to false

- File Processing:
  - Open file using *BufferedReader*
  - Read file line by line
  - Split line into parts `line.split(" ")`
  - Clean words `replaceAll("[^a-zA-Z--]+", "")`
  - Update counters per file
  - Update counters globally

- Real-time GUI
  - Update continuously without blocking application thread
  - Worker thread process files in background and update shared atomic variables
  - No worker thread changes GUI directly
  - *AnimationTimer* is used to refresh GUI once per frame