Project 1: Concurrent File Processing Service

1. Project Overview and Goal

The primary goal of this project is to efficiently analyze a large number of text files by leveraging concurrent processing. This design separates the CPU-intensive file analysis tasks from the I/O-intensive output writing tasks, ensuring high throughput and minimizing bottlenecks.

Functionality:
For every input file, the service calculates:

1. Total number of lines.
2. Total count of unique words (case-insensitive, ignoring punctuation).
3. The text content of the longest line.

The results are collected asynchronously and written to a single output file (analysis_report.txt).

2. Architectural Design: Producer-Consumer Pattern

The system follows a classic **Producer-Consumer** design pattern implemented using Java's java.util.concurrent utilities.

Role	**Component**	**Function**	**Rationale**
Producer (Tasks)	FileProcessor (Worker Threads)	Reads and analyzes a single file. Once analysis is complete, it places the Result object onto the queue.	Decouples analysis from writing. By using a thread pool, we control CPU resource usage and efficiently manage many concurrent tasks.
Shared Buffer	BlockingQueue<Result>	A thread-safe queue (writeQueue) that temporarily holds completed Result objects.	Acts as a buffer between fast producers (FileProcessor) and the single, slower I/O consumer (FileReportWriter). It handles

			synchronization automatically.
Consumer	FileReportWriter (Dedicated Thread)	Continuously takes Result objects from the queue and writes them sequentially to the output file.	Isolates I/O operations to a single thread to prevent file corruption, manage resources efficiently, and ensure results are written in the order they complete (or are ready).
Coordinator	Main.java (Main Thread)	Responsible for discovery of input files, setting up the thread pool, submitting tasks, and managing the graceful shutdown sequence.	Controls the entire application lifecycle.

3. Concurrency Model: Thread Pool and Blocking Queue

Thread Pool (ExecutorService in Main.java)

- **Implementation:** A fixed-size ExecutorService (e.g., Executors.newFixedThreadPool(NUM_THREADS)) is used.
- **Rationale:** File processing is primarily CPU-bound. A thread pool controls the number of active FileProcessor threads, preventing the system from spawning too many threads, which would lead to excessive context switching and reduced performance. The optimal size is typically close to the number of available CPU cores.
- **Task Type:** FileProcessor implements Callable<Result> because it needs to return a value (Result object, even though it also puts the result on the queue) and can potentially throw checked exceptions.

Shared Queue (BlockingQueue<Result>)

- **Implementation:** java.util.concurrent.BlockingQueue (e.g., LinkedBlockingQueue).
- **Rationale:** The BlockingQueue is essential for safe concurrent data transfer.
  - **Safety:** It is inherently thread-safe, meaning we don't need manual synchronized blocks for adding/removing results.
  - **Flow Control:** The take() method in FileReportWriter will **block** until a result is available, efficiently pausing the consumer when no work is ready. The put() method in FileProcessor blocks if the queue is full (which prevents memory overload, though our current implementation uses an unbounded queue for simplicity).

4. Key Component Implementation Details

4.1. FileProcessor.java

Feature	**Implementation Detail**	**Design Rationale**
Resource Management	Uses a **try-with-resources** block for the BufferedReader (try(BufferedReader reader = ...)).	Ensures that the file resource is automatically closed, even if exceptions occur (e.g., IOException).
Word Segmentation	line.toLowerCase().replaceAll("[^a-z0-9\\s]", " ").split("\\s+")	This two-step process is crucial for accurate unique word counting. First, it replaces all punctuation (except spaces) with a space, ensuring words like "cat,dog" are correctly counted as two words ("cat" and "dog"). Then, it splits by one or more spaces (\\s+).
Unique Word Collection	Uses a java.util.HashSet<String>.	The Set guarantees that only unique words are stored. The .size() method (the fixed implementation) efficiently retrieves the count without creating an unnecessary intermediate array.

Result Handling	After processing, the worker explicitly calls writeQueue.put(result) before returning the result.	This ensures that the result is made available to the dedicated writing thread immediately after computation is complete.

4.2. FileReportWriter.java

Feature	Implementation Detail	Design Rationale
Output I/O	Uses PrintWriter wrapped around a standard FileWriter.	PrintWriter provides convenient println() methods for easy text writing. Wrapping it in a try-with-resources block ensures the writer stream is closed automatically on completion/error.
Continuous Loop	The core logic runs inside a while(running) loop, driven by the writeQueue.take().	This allows the writer thread to run continuously in the background, consuming results as soon as they are produced, maintaining real-time output capabilities.
Immediate Write	Includes writer.flush() after every result.toString() write.	Since the results are processed asynchronously and we want immediate visibility, flushing ensures the data is physically written to the file stream/disk without buffering delays.

5. Graceful Shutdown Mechanism: The "Poison Pill"

A core challenge in the Producer-Consumer pattern is knowing when to shut down the consumer thread (FileReportWriter), especially since it indefinitely blocks waiting for data (writeQueue.take()).

- **Mechanism:** The Result.POISON_PILL (a static sentinel value) is used.
- **Flow:**
    1. The Main thread waits for all FileProcessor tasks to complete (by checking the Future objects).
    2. The Main thread shuts down the ExecutorService.
    3. Once the entire system is confident all results have been produced, the Main thread submits the Result.POISON_PILL to the writeQueue.
    4. The FileReportWriter receives the POISON_PILL, prints the final status, sets running = false, and breaks out of its infinite loop, shutting down gracefully.

This prevents the application from hanging indefinitely and guarantees that the writer only terminates after all legitimate analysis results have been processed and written.