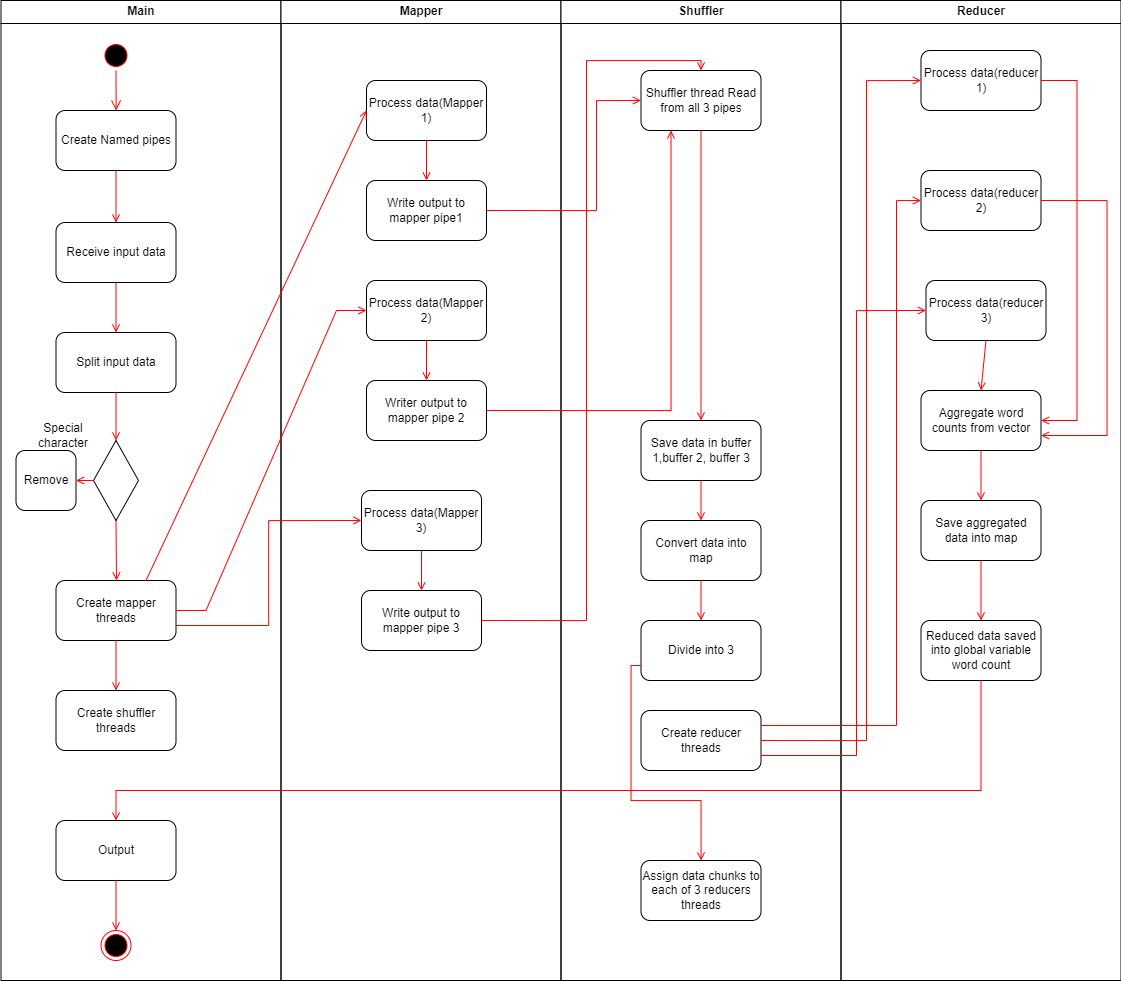
Project Report

22I-1953

22I-1930

22I-1871

## Flow/Activity Diagram:



## Detailed explanation of code:

This code is a multithreaded implementation of a **word count** program using **map-reduce** paradigms. It demonstrates the process of counting the frequency of words from an input sentence, distributed across multiple threads and using inter-process communication (IPC) through pipes.

### **Key Components and Flow:**

1. **Libraries and Setup**:
   * The code includes various headers like <pthread.h>, <map>, and <unistd.h> to enable multithreading, work with data structures, and handle pipes for IPC.
   * The program defines two global mutexes (write\_lock, read\_lock) to synchronize access to shared resources.
   * The global variable word\_count stores the final word counts.
2. **Helper Functions**:
   * **aggregate\_reduced\_data**: This function aggregates word counts from different parts of the program. It adds the word counts from one map into the global word\_count map.
   * **process\_reduction**: Processes a chunk of data (a subset of words and their counts) by summing up the occurrences of each word.
   * **removeSpecialCharacters**: This function sanitizes the input string by removing non-alphanumeric characters (except spaces) to avoid counting unwanted symbols.
3. **Mapper Threads**:
   * Three mapper threads (mapper\_thread1, mapper\_thread2, and mapper\_thread3) handle the word count for different parts of the input data. Each thread:
     + Receives a chunk of the input string as a parameter.
     + Opens a named pipe (PIPE\_1, PIPE\_2, or PIPE\_3) to write the data to.
     + For each word in the chunk, the thread writes a string of the format word,1\n to the pipe, indicating a single occurrence of the word.
     + The threads use a mutex (write\_lock) to ensure that only one thread writes to a pipe at a time.
   * These threads are responsible for the **map** stage of the MapReduce paradigm, where each thread processes its assigned part of the data independently.
4. **Shuffler Thread**:
   * The **shuffler thread** is responsible for reading the word counts from all three pipes (PIPE\_1, PIPE\_2, PIPE\_3).
   * It reads the data into buffers, then parses the word counts, storing them in a map (merged\_data), where the key is the word, and the value is a list of occurrences (one for each count).
   * The shuffler thread simulates the **shuffle** stage of MapReduce, where the distributed word counts from different mappers are aggregated into one collection.
   * After shuffling, the shuffler thread partitions the merged\_data into three smaller chunks and creates three reducer threads to handle these chunks.
5. **Reducer Threads**:
   * The reducer threads (reducer\_thread) take chunks of the shuffled data and process them by summing the word counts.
   * Each reducer processes a subset of the merged\_data and updates the global word\_count map.
   * The reducers handle the **reduce** phase of MapReduce, where the results of the mappers are aggregated into final word counts.
   * Each reducer uses a mutex (write\_lock) to ensure exclusive access to the word\_count map when updating it.
6. **Main Function**:
   * The main function is responsible for setting up the program, initializing pipes, and starting the threads.
   * It splits the input data into three parts, creating three strings (part1\_data, part2\_data, part3\_data).
   * It then creates and joins the mapper threads and the shuffler thread, ensuring synchronization using the pthread\_join function.
   * Finally, the program prints the final aggregated word count from word\_count.
7. **Pipes and Synchronization**:
   * Named pipes (PIPE\_1, PIPE\_2, PIPE\_3) are used for inter-process communication between the mapper threads and the shuffler thread. Each mapper writes its data to a specific pipe, and the shuffler reads from all three pipes.
   * Mutexes (write\_lock and read\_lock) are used to control concurrent access to shared resources, ensuring that only one thread writes to a pipe at a time and that the shuffler thread reads from the pipes after all mappers have finished.

### **Workflow:**

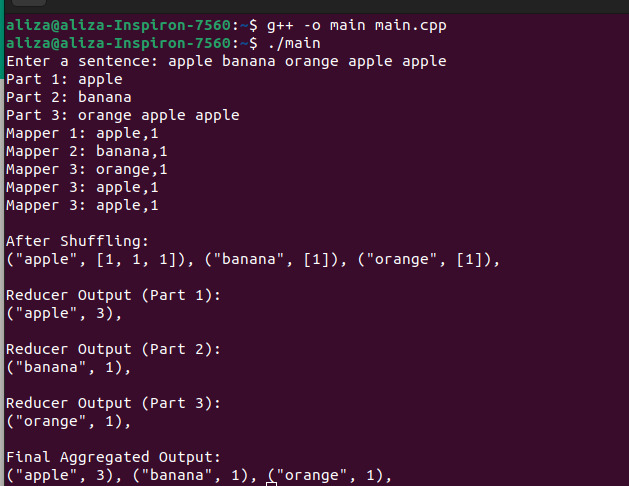
1. The program starts by taking a sentence input from the user and sanitizing it.
2. The input is split into three parts, and each part is processed by one of the mapper threads.
3. Each mapper thread writes the word counts (word,1) to its respective pipe.
4. The shuffler thread reads the data from the pipes, aggregates it, and partitions it into three chunks.
5. Three reducer threads process each chunk of data and aggregate the final word counts.
6. The main function waits for all threads to finish and prints the final word count result.

### **Final Output:**

* The output consists of a list of words along with their corresponding counts, showing how many times each word appeared in the input sentence.

## Test Cases:

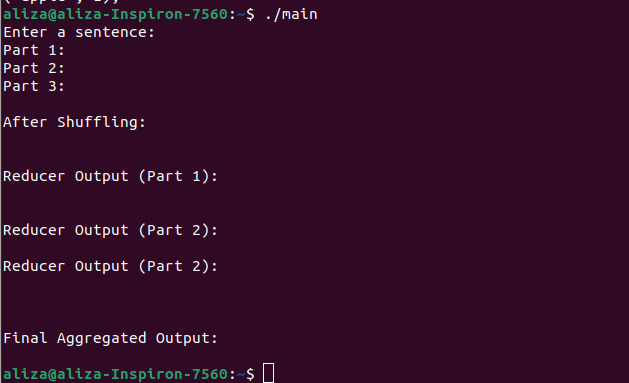
### Test case 1: Basic Input



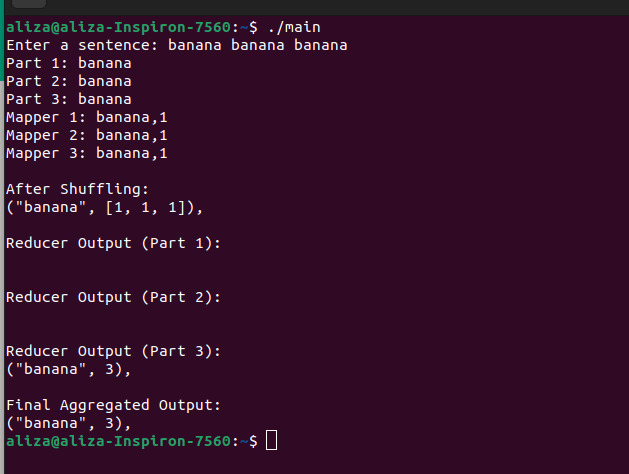
### Test case 2: Single word input



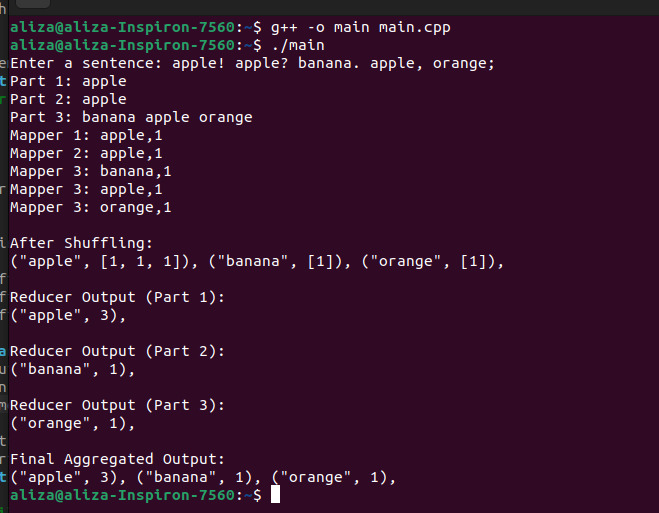
### Test case 3: No word input



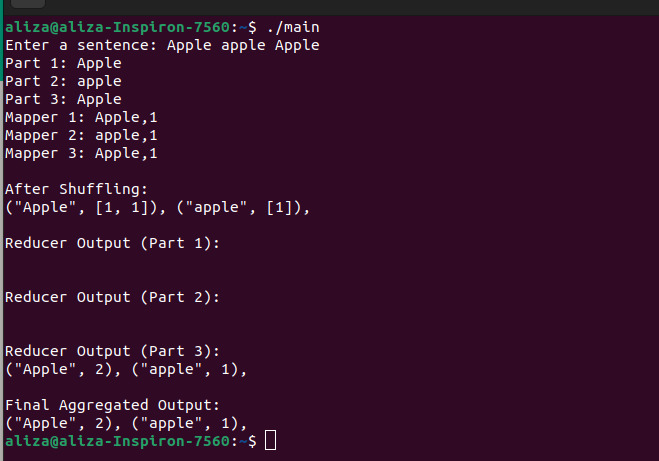
### Test case 4: Repeating same word input



### Test case 5: Special character input



### Test case 6: Case sensitivity



### Test case 7: Long input:

