**Multithreaded Word Counting Program**

**Concept:**

* This program reads a text file and counts the frequency of each word using multiple threads.
* It utilizes **parallel processing** by assigning each sentence segment to a different thread to improve performance.
* To avoid race conditions while updating shared data (wordCount), it uses a **mutex** for thread-safe operations.

**Implementation Steps:**

1. **Read File Content:**
   * The read\_file() function opens and reads the entire contents of a text file into a string.
2. **Segment the Text:**
   * The segment() function splits the input text into segments (sentences), using periods (.) as delimiters.
3. **Launch Threads:**
   * Each text segment is assigned to a separate thread using std::thread.
   * Threads execute the countWords() function.
4. **Process Words in Threads:**
   * Each word in the segment is:
     + Converted to lowercase.
     + Counted in a shared map<string, int> wordCount.
   * A std::mutex (mtx) ensures only one thread updates the map at a time to avoid data corruption.
   * Execution time for each thread is calculated and printed.
5. **Join Threads:**
   * The main thread waits for all threads to complete using join().
6. **Output Results:**
   * After all threads finish, the final word frequencies are printed to the console.

**Observations:**

* **Different Load Distribution:**  
  Since text segments are split based on sentence boundaries, the amount of work per thread may vary .

Source Code

#include <iostream>

#include <fstream>

#include <sstream>

#include <string>

#include <vector>

#include <map>

#include <thread>

#include <mutex>

#include <chrono>

using namespace std;

mutex mtx; // Mutex for thread safety

// Function to process each segment and count word frequencies

void countWords(const string& segment, map<string, int>& wordCount, int threadId) {

auto start = chrono::high\_resolution\_clock::now(); // Start time

stringstream ss(segment);

string word;

while (ss >> word) {

// Process the word (e.g., convert to lowercase, remove punctuation)

for (char& ch : word) {

ch = tolower(ch);

}

// Increment word count

{

lock\_guard<mutex> lock(mtx); // Lock the mutex to ensure thread-safe access

wordCount[word]++;

}

}

auto end = chrono::high\_resolution\_clock::now(); // End time

chrono::duration<double> duration = end - start;

cout << "Thread " << threadId << " execution time: " << duration.count() << " seconds." << endl;

}

vector<string> segment(const string& text) {

vector<string> segments;

string current\_segment = "";

for (char c : text) {

current\_segment += c; // Keep adding characters to the string

if (c == '.') {

segments.push\_back(current\_segment);

current\_segment = ""; // Reset the string for the next segment

}

}

if (!current\_segment.empty()) {

segments.push\_back(current\_segment);

}

return segments;

}

string read\_file(const string& file\_name) {

ifstream file(file\_name);

if (!file) {

cerr << "Error: File could not be opened!" << endl;

return "";

}

// Use stringstream to read the file content

stringstream read;

read << file.rdbuf(); // Read the entire file content

return read.str(); // Return the content as a string

}

int main() {

string text = read\_file("C:\\Users\\salla\\OneDrive\\Desktop\\Text Parsing\\Text.txt"); // Read the file content

vector<string> segments = segment(text);

// Shared map to store word frequencies that the threads will work on

map<string, int> wordCount;

// Vector to hold threads

vector<thread> threads;

// Create and launch a thread for each segment

for (int i = 0; i < segments.size(); ++i) {

threads.push\_back(thread(countWords, ref(segments[i]), ref(wordCount), i + 1));

}

// Join all threads (wait for all of them to finish)

for (thread& t : threads) {

t.join();

}

// Print the word frequencies

cout << "Word frequencies:" << endl;

for (const auto& pair : wordCount) {

cout << pair.first << ": " << pair.second << endl;

}

return 0;

}