**Task 2**

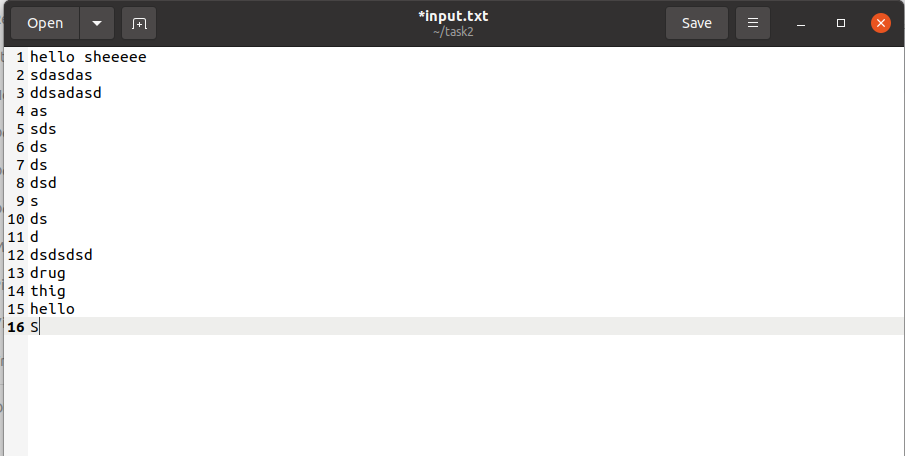
**Description**

This task required a multi-threaded approach to count frequency of words in a text file, the file was divided into N segments, each assigned to a different thread to utilize parallel processing. The main thread waits for all the worker threads to finish then combines the result into a final frequency-count, this makes the program much more efficient since it’s using parallel programming.

**Code**

#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#include <pthread.h>  
#include <ctype.h>  
  
#define MAX\_WORDS 1000  
#define MAX\_WORD\_LEN 100  
  
typedef struct {  
    char word[MAX\_WORD\_LEN];  
    int count;  
} WordFreq;  
  
typedef struct {  
    char \*segment;  
    int segment\_size;  
    WordFreq \*word\_freq;  
    int word\_count;  
} ThreadData;  
  
void \*count\_word\_frequency(void \*arg) {  
    ThreadData \*data = (ThreadData \*)arg;  
    char \*segment = data->segment;  
    int segment\_size = data->segment\_size;  
    WordFreq \*word\_freq = data->word\_freq;  
    int \*word\_count = &data->word\_count;  
  
    char word[MAX\_WORD\_LEN];  
    int word\_len = 0;  
  
    for (int i = 0; i < segment\_size; i++) {  
        if (isalpha(segment[i]) || segment[i] == '\'') {  
            word[word\_len++] = tolower(segment[i]);  
        } else if (word\_len > 0) {  
            word[word\_len] = '\0';  
            word\_len = 0;  
  
            int found = 0;  
            for (int j = 0; j < \*word\_count; j++) {  
                if (strcmp(word\_freq[j].word, word) == 0) {  
                    word\_freq[j].count++;  
                    found = 1;  
                    break;  
                }  
            }  
  
            if (!found) {  
                strcpy(word\_freq[\*word\_count].word, word);  
                word\_freq[\*word\_count].count = 1;  
                (\*word\_count)++;  
            }  
        }  
    }  
  
    if (word\_len > 0) {  
        word[word\_len] = '\0';  
  
        int found = 0;  
        for (int j = 0; j < \*word\_count; j++) {  
            if (strcmp(word\_freq[j].word, word) == 0) {  
                word\_freq[j].count++;  
                found = 1;  
                break;  
            }  
        }  
  
        if (!found) {  
            strcpy(word\_freq[\*word\_count].word, word);  
            word\_freq[\*word\_count].count = 1;  
            (\*word\_count)++;  
        }  
    }  
  
    pthread\_exit(NULL);  
}  
  
void merge\_word\_frequencies(WordFreq \*final\_freq, int \*final\_count, WordFreq \*thread\_freq, int thread\_count) {  
    for (int i = 0; i < thread\_count; i++) {  
        int found = 0;  
        for (int j = 0; j < \*final\_count; j++) {  
            if (strcmp(final\_freq[j].word, thread\_freq[i].word) == 0) {  
                final\_freq[j].count += thread\_freq[i].count;  
                found = 1;  
                break;  
            }  
        }  
        if (!found) {  
            strcpy(final\_freq[\*final\_count].word, thread\_freq[i].word);  
            final\_freq[\*final\_count].count = thread\_freq[i].count;  
            (\*final\_count)++;  
        }  
    }  
}  
  
int find\_word\_boundary(char \*text, int position, int max\_size) {  
    for (int i = position; i < max\_size; i++) {  
        if (isspace(text[i]) || text[i] == '\0') {  
            return i + 1;  
        }  
    }  
    return max\_size;  
}  
  
int main(int argc, char \*argv[]) {  
    if (argc != 3) {  
        fprintf(stderr, "Usage: %s <filename> <N>\n", argv[0]);  
        exit(EXIT\_FAILURE);  
    }  
  
    char \*filename = argv[1];  
    int N = atoi(argv[2]);  
  
    FILE \*file = fopen(filename, "r");  
    if (!file) {  
        perror("Failed to open file");  
        exit(EXIT\_FAILURE);  
    }  
  
    fseek(file, 0, SEEK\_END);  
    long file\_size = ftell(file);  
    fseek(file, 0, SEEK\_SET);  
  
    char \*file\_content = (char \*)malloc(file\_size + 1);  
    fread(file\_content, 1, file\_size, file);  
    file\_content[file\_size] = '\0';  
    fclose(file);  
  
    pthread\_t threads[N];  
    ThreadData thread\_data[N];  
    WordFreq final\_freq[MAX\_WORDS];  
    int final\_count = 0;  
  
    int segment\_size = file\_size / N;  
    int start = 0;  
  
    for (int i = 0; i < N; i++) {  
        int end = (i == N - 1) ? file\_size : find\_word\_boundary(file\_content, start + segment\_size, file\_size);  
  
        thread\_data[i].segment = file\_content + start;  
        thread\_data[i].segment\_size = end - start;  
        thread\_data[i].word\_freq = (WordFreq \*)malloc(MAX\_WORDS \* sizeof(WordFreq));  
        thread\_data[i].word\_count = 0;  
  
        pthread\_create(&threads[i], NULL, count\_word\_frequency, &thread\_data[i]);  
        start = end;  
    }  
  
    for (int i = 0; i < N; i++) {  
        pthread\_join(threads[i], NULL);  
    }  
  
    for (int i = 0; i < N; i++) {  
        merge\_word\_frequencies(final\_freq, &final\_count, thread\_data[i].word\_freq, thread\_data[i].word\_count);  
        free(thread\_data[i].word\_freq);  
    }  
  
    for (int i = 0; i < final\_count; i++) {  
        printf("%s: %d\n", final\_freq[i].word, final\_freq[i].count);  
    }  
  
    free(file\_content);  
  
    return 0;  
}

**Figures**



**Github link**

[**https://github.com/FirasAhmed2/Operating-systems-coursework.git**](https://github.com/FirasAhmed2/Operating-systems-coursework.git)