UNIVERSITY PARTNER

UNIVERSITY OF WOLVERHAMPTON

HERALD COLLEGE KATHMANDU

6CS005 High Performance Computing

Portfolio 1 : Task1

Student Name: Pratik Dhimal

University ID: 2407779

Group: L6CG1

Tutor: Yamu Poudel

Submission Date: 20th December, 2025

Table of Contents

1. Introduction

The single-threaded frequency counter of words is not efficient with increase of file size; at some point, the frequency counter takes more time to run and the CPU is not fully utilized. In order to solve this issue, parallel programming programs may be resorted to in order to split the workset into several threads, where various parts of the input file are being processed concurrently. The given project is an implementation of a parallel word frequency counter written in POSIX threads (pthreads) in the C programming language that proves that concurrency may considerably enhance the speed without compromising the correctness.

The program operates by loading the entire input file into a shared memory buffer, and cutting this buffer into several slices and putting each slice into its own thread to process. Words are removed by each thread, based on its allocated part of the buffer and a global word-frequency data structure is updated in a thread-safe fashion through mutex locks. To deal with boundary conditions in which a word can be cross-thread (cut into two slices) a special care is given to make sure that words are counted only once. The result is a list of sorted words and their frequency, it is written into an output file. This methodology lays emphasis on major ideas in parallel execution, synchronization, memory, and edge-case in multithreaded systems.

2. Execution

2.1. Input Validiation

This programm expects 2 additional argument which are the fiel containing the word and another is the number of threads to use. In this very first steps we check weather or there are right number of arguments which is 3 in this case, and also we check and notify the user if the number of thread is a positve integer. As shown in **Figure 1.**

2.2. File handeling

In this process we read the file in the memory. If the file doesn't exist than a error message is shown to the user. Initially the file pointer as shown in **Figure 2** is pointing to the start of the file than we use the **fseek()** function with **SEEK_END** argument to change the pointer to the end and then read the current pointer value using **ftell()** which return the **file size** in bytes. Next we use **rewind()** with the file pointer argument to repoint the pointer to the start of the file. Now we check weather or not the file is empty using the **file_size,** if it is zero than the file is empty. If the file is empty we close the programm. If file content exist than we use **malloc()** to allocate memory. We allocate one additioal byte of memory which is later filled with end terminator '\0' to indicate the end of cotnet. Lastly we fill the memory with the **file data** usign the **fread()** function and close the file using **fclose()** function.

2.3. Buffer slicing and thread creation

Here the thread id is initialized and the thread argument as shown in **Figure 4** is also declared using malloc as shown in **Figure 3**. Next a loop is runned to the thread count so that we can populate the thread arguments with necessary value. As shown in **Figure 5** we fill each thread arguments with the start(size from where to start in bytes), end(size where to end in bytes), buffer(the actual file buffer), and the total buffer size(bytes). Next we run the thread function using the **pthread_create().** In this case we run the **process_slice()** function with the thread arguments creted previously.Lastly we joint all the threads using the **pthred_join()** function. The execution of **process _slice()** function is shown **below.**

2.4. Result Generation and Output Handling

As shown in **Figure 6**, the final processing of results, reporting, and cleanup of all threads that have completed counting words are the responsibilities of this part of the program. To begin with, the global words array as shown in **Figure 4** is sorted using **qsort()**in a descending order of frequency of the words in the array where the most frequently appearing words come at the beginning of the output. The program subsequently opens an output file with the error control so as to check if the file result.txt has been created successfully. A structured header is appended to the file alongside the number of unique words which are summed up and the column names that should be read. The program repeats the sorted word list reading and writing out each word and its frequency to the file and at the same time adds together the total number of words handled by adding individual counts together. The key statistics are also shown on the console afterward after printing a summary section (total words processed) so that the user can give feedback. The last thing that happens is the program closes the output file, the dynamically allocated memory (buffer, thread array, and thread argument array) are freed, and 0 is returned indicating successful execution.

2.5. The process_slice() function

As shown in **Figure 7,** this is the worker logic that is the essence of every thread and that which is used to process a given segment of the input text buffer. Each thread is provided with a **ThreadArgs** structure that has the shared text buffer and start and end indices of it slice. The operation starts with the initialisation of a temporary **current_word buffer** and a position index with which words are constructed character by character. As a way of avoiding counting a partial word, the thread initially examines whether the index at which it currently is in an **alpha-numeric sequence** here **the non alpha-numberic sequenc is the "\n"** which indicated next line is in the middle of the sequence; in that case, the index is advanced until the index reaches a word boundary. This loop then reads characters in the slice assigned, adding alphanumeric characters to the current word, and changing them to lowercase to enforce the counting of cases being insensitive to the case. Upon coming to a non-alphanumeric character, the word that has been formed is terminated at the end and safely inserted into the global word list.

The function also takes precautions with the boundary conditions in the end of the slice where a word can be longer than the range assigned to the thread. Once the initial loop is done, another loop starts again reading characters beyond the slice point until a **delimiter** is located, which makes sure that a word cut between two slices is counted once by the current thread. Should there be a meaningful word in the buffer at the end of this process then it is completed and inserted into the world collection. This functionality ensures that word counts are made correctly without redundancy or data races by making sure that the index adjustment is done very carefully, word construction is done very carefully, and updates are synchronized even when multiple threads are using the same shared buffer.

2.6. The add_word() function

This function performs the work of safely updating the data structure of words frequencies in the world in a multithreaded setting. When a thread recognizes a complete word it calls **add_word()**, which initially takes out a **mutex lock** to be able to exclusively access the shared words array and variable word count to avoid **race conditions**. The function then conducts a linear search of the available entries in order to determine whether the word already exists; in this case of a match the appropriate frequency count is increased and the mutex returned. When the word is not found and there was still space, the feature will add the new word to the array, set its count to one, and the total number of unique words. Lastly, **the mutex is released** and only a single thread is allowed to update the shared data at any point in time and keeps the concurrent execution **correct and consistent**.

3. Conclusion

In summary, this word counting job managed to prove the effectiveness of a multi-threaded way of text processing through the slicing of a file and processing them simultaneously through safe mutex synchronization. The program correctly deals with boundary of words between thread slices, case insensitive counting and race conditions on updating the global word frequency list. The implementation creates meaningful output by sorting the final results by their descending frequency, which demonstrates better performance on large files and the idea supported the main concepts of concurrency, synchronization, and file processing in parallel in C with POSIX threads.

4. Appendix

```
int main(int argc, char *argv[]) {
    if (argc != 3) {
        printf("Usage: %s <filename> <thread_count>\n", argv[0]);
        return 1;
    }

    char *endptr;
    long thread_count = strtol(argv[2], &endptr, 10);
    if (*endptr != '\0' || thread_count < 1) {
        printf("Please enter a valid positive integer for the thread count.\n");
        return 1;
    }
```

Figure 1 Input

```
FILE *file = fopen(argv[1], "r");
if (file == NULL) {
    perror("Error opening input file");
    return 1;
}

fseek(file, 0, SEEK_END);
long file_size = ftell(file);
rewind(file);

if (file_size == 0) {
    printf("Error: File is empty\n");
    fclose(file);
    return 1;
}

char *buffer = malloc(file_size + 1);
if (buffer == NULL) {
    perror("Error allocating memory");
    fclose(file);
    return 1;
}

size_t bytes_read = fread(buffer, 1, file_size, file);
if (bytes_read == 0) {
    perror("Error reading file");
    free(buffer);
    fclose(file);
    return 1;
}
buffer[bytes_read] = '\0';
fclose(file);
```

Figure 2 File handeling

```
pthread_t *threads = malloc(thread_count * sizeof(pthread_t));
ThreadArgs *thread_args = malloc(thread_count * sizeof(ThreadArgs));

if (threads == NULL || thread_args == NULL) {
    perror("Error allocating memory for threads");
    free(buffer);
    free(threads);
    free(thread_args);
    return 1;
}

long slice_size = actual_size / thread_count;
```

Figure 3 threads argument creation

```
typedef struct {
    char word[50];
    int count;
} WordEntry;

WordEntry words[5000];
int word_count = 0;

pthread_mutex_t lock;

typedef struct {
    char *text_buffer;
    long start_index;
    long end_index;
    long buffer_size;
} ThreadArgs;
```

Figure 4 Structures

```
for (int i = 0; i < thread_count; i++) {
    thread_args[i].text_buffer = buffer;
    thread_args[i].start_index = i * slice_size;
    thread_args[i].end_index = (i == thread_count - 1) ? actual_size : (i + 1) * slice_size;
    thread_args[i].buffer_size = actual_size;

    if (pthread_create(&threads[i], NULL, process_slice, &thread_args[i]) != 0) {
        perror("Failed to create thread");
        free(buffer);
        free(threads);
        free(thread_args);
        return 1;
    }
}

for (int i = 0; i < thread_count; i++) {
    pthread_join(threads[i], NULL);
}
```

Figure 5 Pouplating Threads args and running threads

```
// Sort words by frequency (descending order)
qsort(words, word_count, sizeof(WordEntry), compare_frequency);

FILE *out = fopen("result.txt", "w");
if (out == NULL) {
    perror("Error opening output file");
    free(buffer);
    free(threads);
    free(thread_args);
    return 1;
}

fprintf(out, "Word Frequency Count Results\n");
fprintf(out, "==============================\n");
fprintf(out, "Total unique words: %d\n", word_count);
fprintf(out, "(Sorted by frequency - descending order)\n\n");
fprintf(out, "%-20s %s\n", "Word", "Count");
fprintf(out, "-------------------- ------\n");

int total_words = 0;
for (int i = 0; i < word_count; i++) {
    fprintf(out, "%-20s %6d\n", words[i].word, words[i].count);
    total_words += words[i].count;
}

fprintf(out, "\n==============================\n");
fprintf(out, "Total words processed: %d\n", total_words);

printf("Processing complete!\n");
printf("Total unique words: %d\n", word_count);
printf("Total words processed: %d\n", total_words);
printf("Results saved to result.txt\n");

fclose(out);
free(buffer);
free(threads);
free(thread_args);

return 0;
```

Figure 6 Adding content to file and shwoign necessary logs

```
void *process_slice(void *arg) {
    ThreadArgs *data = (ThreadArgs *)arg;
    if (data == NULL) return NULL;

    char current_word[50];
    int pos = 0;

    long i = data->start_index;

    // This line Skips partial word at the start (if we're in the middle of a word)
    if (i > 0 && isalnum(data->text_buffer[i-1])) {
        while (i < data->end_index && i < data->buffer_size && isalnum(data->text_buffer[i])) {
            i++;
        }
    }

    // Process words in this slice
    for (; i < data->end_index && i < data->buffer_size; i++) {
        char c = data->text_buffer[i];
        if (isalnum(c)) {
            if (pos < 49) {
                current_word[pos++] = tolower(c);
            }
        } else {
            if (pos > 0) {
                current_word[pos] = '\0';
                add_word(current_word);
                pos = 0;
            }
        }
    }

    // Complete the word that crosses the boundary
    while (i < data->buffer_size && isalnum(data->text_buffer[i])) {
        if (pos < 49) {
            current_word[pos++] = tolower(data->text_buffer[i]);
        }
        i++;
    }

    if (pos > 0) {
        current_word[pos] = '\0';
        add_word(current_word);
    }

    return NULL;
}
```

Figure 7 Process slice function

```
void add_word(char *w) {
    pthread_mutex_lock(&lock);
    for (int i = 0; i < word_count; i++) {
        if (strcmp(words[i].word, w) == 0) {
            words[i].count++;
            pthread_mutex_unlock(&lock);
            return;
        }
    }
    if (word_count < 5000) {
        strcpy(words[word_count].word, w);
        words[word_count].count = 1;
        word_count++;
    }
    pthread_mutex_unlock(&lock);
}
```

Figure 8 The add word function