Faculty of Engineering and Technology
Electrical and Computer Engineering Department

Computer Design Lab
ENCS4110

Project 1

Prepared by:
Basmala abu hakema 1220184

Instructor: **Yazan Abu Farha**

Section: **1**
Date: **20-11-2024**

Table of Contents

Table of figures:

System Environment:

1. Computer Specifications:

- **Processor (Cores and Speed):**

My system is equipped with an **Intel Core i7-12700H** processor, which has 14 cores and 20 logical processors, running at a base clock speed of **2.3 GHz**.

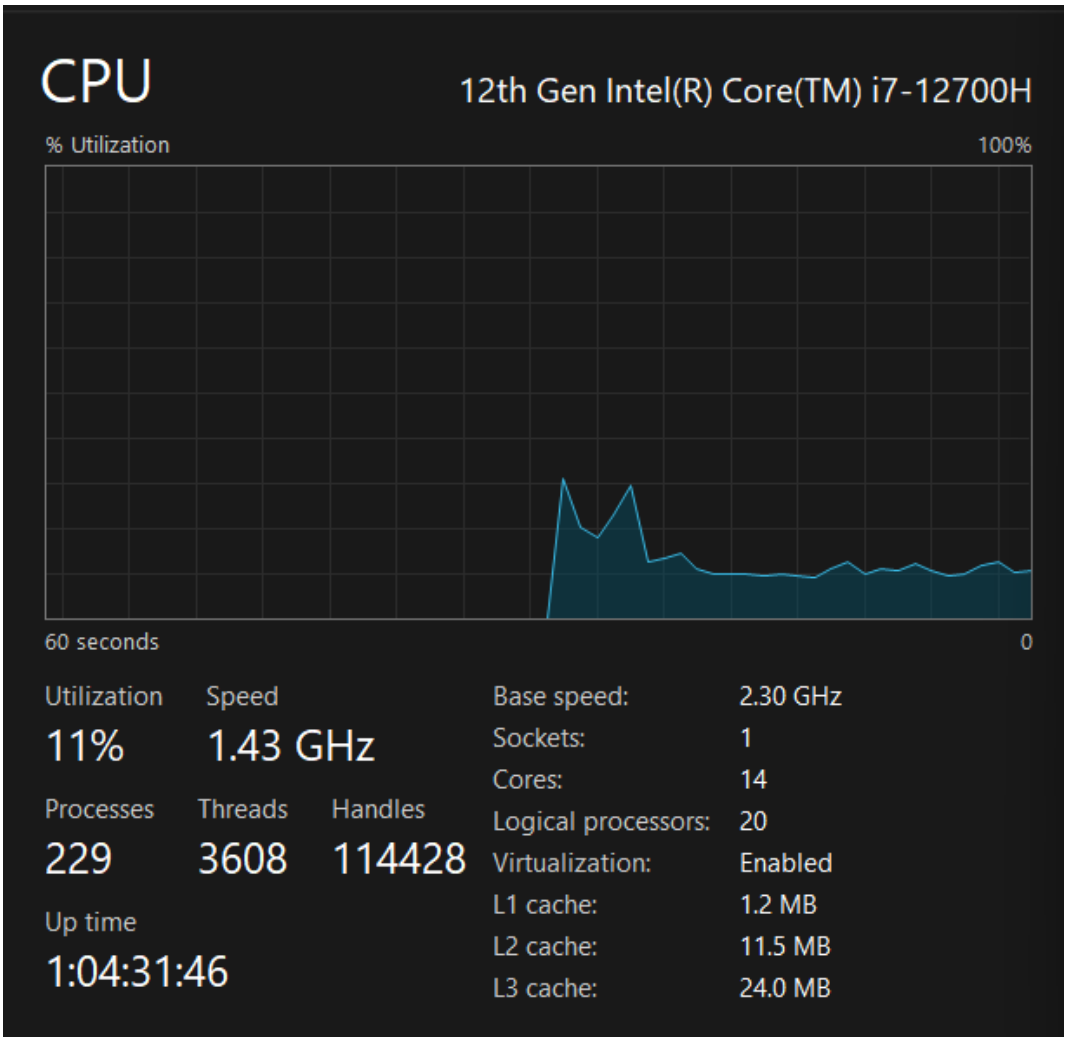

Figure 1: CPU

- **Memory (RAM):**

The system has **16 GB** of **DDR4 RAM** running at **3200 MHz**.

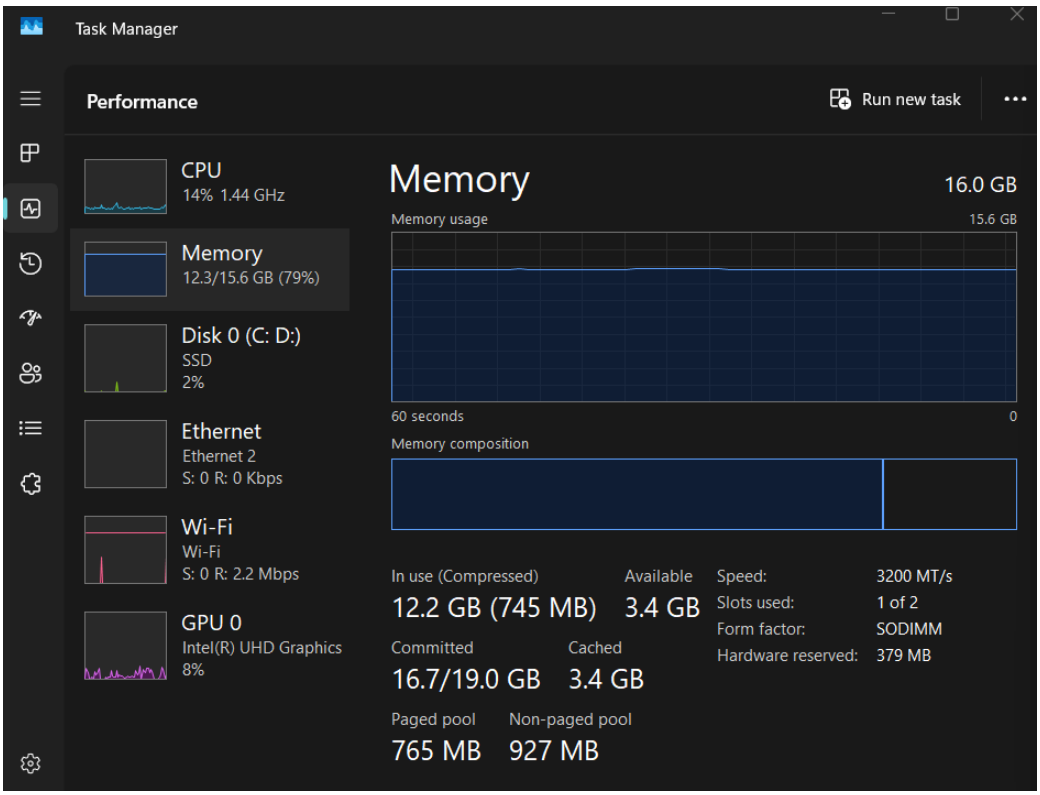

Figure 2: memory

- **Storage:**

The primary storage is a **1 TB SSD** (Solid State Drive) with a read/write speed of up to **550 MB/s**. This ensures fast data access for large projects and applications.

- **Graphics (GPU):**

It uses an integrated **Intel(R) UHD Graphics 630**.

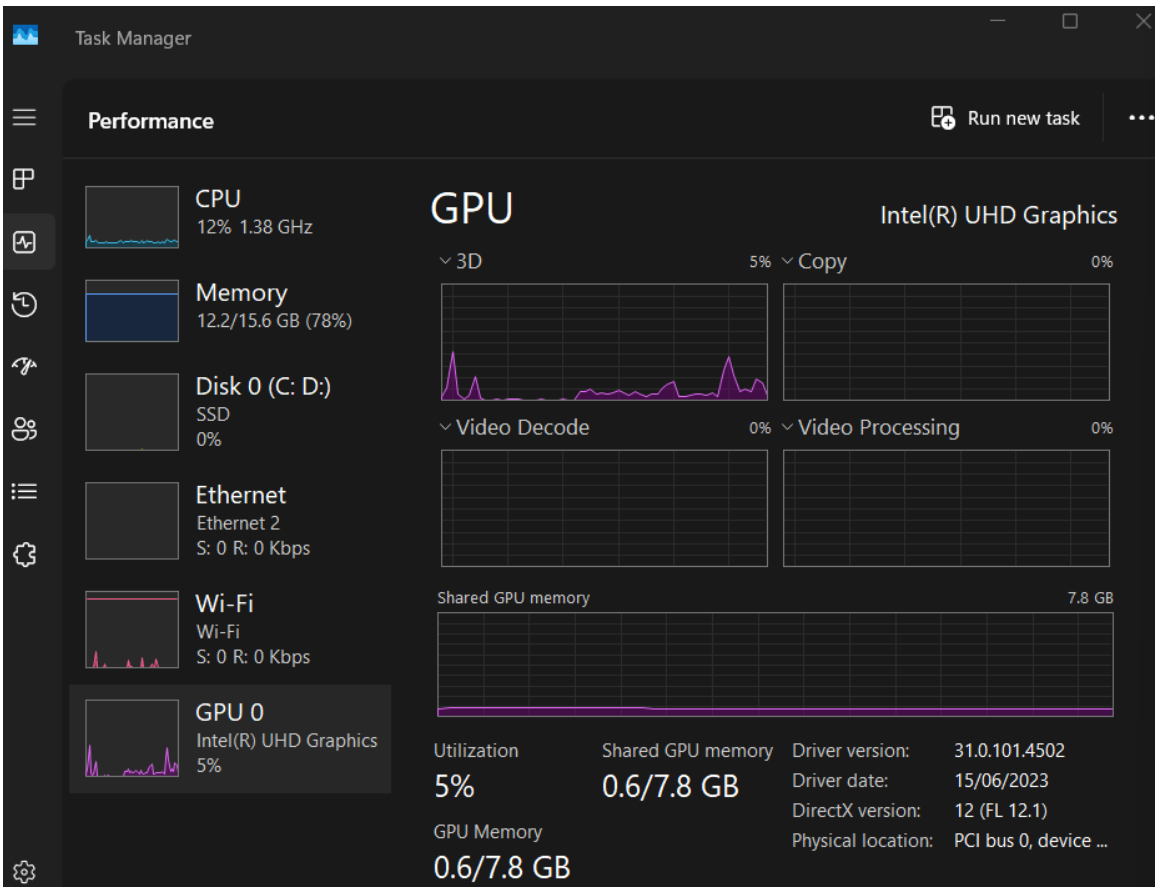

Figure 3: GPU

2. Operating System:

I developed and ran the project on **Windows 10 Pro (64-bit)**. This OS provides a robust platform for development with full support for C compilers and integrated development environments (IDEs).

3. Programming language:

- The project was implemented using **C**, a widely used systems programming language known for its performance and low-level memory manipulation capabilities.
- **C** was chosen for this project due to its efficiency, speed, and direct control over system resources, which made it ideal for low-level programming tasks, performance-critical applications, or hardware-level interfacing.

4. Integrated Development Environment (IDE) and Tools:

- **IDE Used:**

I used **Gedit**, the default text editor in Ubuntu, for writing the C program. Gedit is a lightweight and user-friendly text editor with syntax highlighting, making it suitable for coding in C. Though it's simple, it provides an efficient environment for small and medium-sized projects. Additionally, I used the terminal for compiling and running the code, leveraging Ubuntu's native support for C development.

- **Compiler:**

The project was compiled using the **GNU GCC Compiler**, which is pre-installed on Ubuntu. GCC is highly efficient for compiling C code, and it produces optimized, executable files that are well-suited for Linux-based systems.

5. Virtual Machine Usage:

the project was developed within a **Ubuntu virtual machine** created using **Oracle VirtualBox**. This setup provided an isolated development environment, simulating a Linux environment on my physical machine. The code was written, compiled, and executed within the Ubuntu VM to ensure compatibility and a consistent development environment.

Achieving Multiprocessing and Multithreading:

In this project, I utilized both **multiprocessing** and **multithreading** to enhance performance and allow concurrent execution of tasks. Below are the APIs and functions I used to achieve this:

1. Naive approach:

The naive approach to parallel computing involves executing tasks in a sequential manner, with each task being performed one after the other within a single thread or process. This method is simple to implement and can work well for small or uncomplicated tasks where the complexity of parallelism isn't needed. However, since tasks are executed in sequence, there is no performance improvement, making this approach less effective for computationally heavy tasks that could benefit from parallel execution.

code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#define MAX_LENGTH 50 // Maximum length of each word
#define MAX_WORDS 17500000 // Maximum number of words to read from the file

// Define a structure to store a word and its frequency count
typedef struct {
    char word[MAX_LENGTH]; // The word itself
    int count;             // The number of occurrences of the word
} WordCount;

// Function to read words from the file and store them in an array
int readWords(const char *filename, char **words, int maxWords) {
    FILE *file = fopen(filename, "r"); // Open the file in read mode
    if (file == NULL) {
        printf("Error opening file.\n");
        return -1; // Return an error if file cannot be opened
    }

    int wordCount = 0;
    // Read words from the file until we reach the maximum or end of the file
    while (wordCount < maxWords && fscanf(file, "%99s", words[wordCount]) == 1) {
        wordCount++; // Increment the word count for each word read
    }

    if (wordCount >= maxWords) {
        printf("Reached maximum word count (%d).\n", maxWords); // Warn if maximum words are reached
    }
```

I defined a struct to Represents a word and its frequency.

Read_words function (additional function to Reads words from the text8.txt file and stores them in the words array.)

```
    fclose(file); // Close the file after reading
    return wordCount; // Return the total number of words read
}

// Function to process the array of words, calculate frequencies, and store them in another array
int processWords(char **words, int totalWords, WordCount *wordCounts, int maxWords) {
    int wordCount = 0;

    // Iterate through all the words
    for (int i = 0; i < totalWords; i++) {
        // Check if the word is already in the wordCounts array
        int found = 0;
        for (int j = 0; j < wordCount; j++) {
            if (strcmp(wordCounts[j].word, words[i]) == 0) {
                wordCounts[j].count++; // Increment the count if the word is found
                found = 1;
                break;
            }
        }

        // If the word is not found, add it to the wordCounts array
        if (!found) {
            if (wordCount >= maxWords) {
                printf("Exceeded maximum unique word count.\n");
                return wordCount; // Return if maximum word count is exceeded
            }
            strcpy(wordCounts[wordCount].word, words[i]); // Store the new word
            wordCounts[wordCount].count = 1; // Initialize its count
            wordCount++; // Increment the unique word count
        }
    }
```

ProcessWords function: Calculates the frequency of each word and stores the results in the wordCounts array.

```
    return wordCount; // Return the total number of unique words
}

// Comparison function for qsort to sort the wordCounts array by frequency (descending order)
int compare(const void *a, const void *b) {
    return ((WordCount *)b)->count - ((WordCount *)a)->count; // Sort in descending order by frequency
}

// Function to list the top 10 most frequent words
void listTopWords(WordCount *wordCounts, int wordCount) {
    qsort(wordCounts, wordCount, sizeof(WordCount), compare); // Sort the words by frequency

    printf("Top 10 most frequent words:\n");
    for (int i = 0; i < 10 && i < wordCount; i++) {
        printf("%s: %d\n", wordCounts[i].word, wordCounts[i].count); // Print the top 10 words with their counts
    }
}

int main(void) {
    clock_t start, end;
    double time_spent;

    start = clock(); // Start the clock to measure execution time

    // Dynamically allocate memory for word storage
    char **words = (char **)malloc(MAX_WORDS * sizeof(char *)); // Allocate memory for word pointers
    if (words == NULL) {
        printf("Memory allocation failed for words.\n");
        return 1; // Exit if memory allocation fails
    }
```

List_Top_Words function: Sorts the wordCounts array by frequency and prints the top 10 most frequent words.
Compare function: Comparison function for qsort. Orders WordCount elements in descending order of frequency.

Main function: Memory Allocation for Words: Dynamically allocates memory for storing words. Allocates a MAX_LENGTH buffer for each word in the words array.

```c
    // Allocate memory for each word
    for (int i = 0; i < MAX_WORDS; i++) {
        words[i] = (char *)malloc(MAX_LENGTH * sizeof(char)); // Allocate memory for each word
        if (words[i] == NULL) {
            printf("Memory allocation failed for word %d.\n", i);
            return 1; // Exit if memory allocation fails
        }
    }

    // Read words from the file and store them in the 'words' array
    int totalWords = readWords("C:/Users/Asus/Downloads/text8.txt", words, MAX_WORDS);
    if (totalWords == -1) {
        return 1; // Exit if reading the file fails
    }

    // Dynamically allocate memory for word frequencies
    WordCount *wordCounts = (WordCount *)malloc(MAX_WORDS * sizeof(WordCount));
    if (wordCounts == NULL) {
        printf("Memory allocation failed for wordCounts.\n");
        return 1; // Exit if memory allocation fails
    }

    // Process the words to count their frequencies
    int wordCount = processWords(words, totalWords, wordCounts, MAX_WORDS);

    // List the top 10 most frequent words
    listTopWords(wordCounts, wordCount);
```

Read Words: Reads words from text8.txt into the words array. Prints the total number of words read.

Memory Allocation for Word Counts: Allocates memory for storing word frequencies.

Process Words: Calculates word frequencies and stores them in wordCounts.

List Top Words: Sorts and prints the top 10 most frequent words.

```c
    // List the top 10 most frequent words
    listTopWords(wordCounts, wordCount);

    // Free dynamically allocated memory
    for (int i = 0; i < MAX_WORDS; i++) {
        free(words[i]); // Free each word
    }
    free(words); // Free the words array
    free(wordCounts); // Free the word counts array

    end = clock(); // End the clock
    time_spent = (double)(end - start) / CLOCKS_PER_SEC; // Calculate execution time
    printf("EXECUTION TIME: %f seconds --- FIXED APPROACH ---\n", time_spent);

    return 0;
}
```

Figure 4: code for naïve approach

Free Memory: Frees all dynamically allocated memory to prevent memory leaks.

Execution Time: prints and calculates the total execution time

Figure 5: output of naive

Discussion:

This C program reads a large text file, processes its words, and calculates the frequency of each unique word. It begins by dynamically allocating memory for storing up to MAX_WORDS words and their corresponding frequencies. The program reads the words from the text8.txt file using the readWords() function, storing them in an array. It then counts the frequency of each word by comparing each new word with the already stored words in the processWords() function. After processing, the program sorts the words by frequency using qsort() and displays the top 10 most frequent words. The execution time is measured using clock(), and at the end of the program, all dynamically allocated memory is freed to prevent memory leaks. This approach ensures that the program can handle large amounts of data efficiently while maintaining readability and clear structure. It took about 10 mins to execute.

2. Multiprocessing approach:

A multiprocessing operating system is defined as a type of operating system that makes use of more than one CPU to improve performance. Multiple processors work parallelly in multi-processing operating systems to perform the given task. All the available processors are connected to peripheral devices, computer buses, physical memory, and clocks. The main aim of the multi-processing operating system is to increase the speed of execution of the system. The use of a multiprocessing operating system improves the overall performance of the system. For example, UNIX, LINUX, and Solaris are the most widely used multi-processing operating system.

Code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/mman.h>
#include <time.h>

#define MAX_TERMS 17500000  // Maximum number of unique terms
#define WORD_LIMIT 50       // Maximum length of a term (word)
#define PROCESS_COUNT 8     // Number of processes to divide the workload
#define LOCAL_STORAGE 1000000  // Local storage per process for term frequencies

// Structure to store term and its frequency
typedef struct {
    char term[WORD_LIMIT];
    int frequency;
} TermFrequency;

// Shared memory pointers for global terms and their count
TermFrequency *globalTerms;
int *globalTermCount;

// Update term frequency in the local buffer
void update_term(TermFrequency *buffer, int *bufferCount, const char *term) {
    // Search for the term in the buffer
    for (int i = 0; i < *bufferCount; i++) {
        if (strcmp(buffer[i].term, term) == 0) {
            buffer[i].frequency++;  // Increment frequency if term exists
            return;
        }
    }
    // Add new term if not found
    strcpy(buffer[*bufferCount].term, term);
    buffer[*bufferCount].frequency = 1;
    (*bufferCount)++;
}
```

Here I defined a struct to Stores each term and its frequency.

And defined globalTerms: Shared memory for storing terms and their frequencies.
globalTermCount: Shared memory pointer for tracking the number of stored terms.

Update_ term function: Updates or adds a term in the local buffer.

```
38
39 // Analyze a segment of the file
40 void analyze_segment(const char *filepath, int start, int finish, TermFrequency *buffer, int *bufferCount) {
41     FILE *data = fopen(filepath, "r");
42     if (!data) {
43         perror("Failed to open file");
44         exit(1);
45     }
46
47     fseek(data, start, SEEK_SET);
48
49     // Skip partial word if not at word boundary
50     if (start != 0) {
51         char c;
52         while (fread(&c, 1, 1, data) && (c != ' ' && c != '\n')) {
53             // Skip partial word
54         }
55     }
56
57     char term[WORD_LIMIT];
58     // Read terms within the segment and update their frequency
59     while (ftell(data) < finish && fscanf(data, "%49s", term) != EOF) {
60         update_term(buffer, bufferCount, term);
61     }
62
63     fclose(data);
64 }
65
66 // Integrate results from the local buffer into the global shared array
67 void integrate_results(TermFrequency *buffer, int bufferCount) {
68     for (int i = 0; i < bufferCount; i++) {
69         int exists = 0;
70         // Check if the term exists in the global shared array
71         for (int j = 0; j < *globalTermCount; j++) {
72             if (strcmp(globalTerms[j].term, buffer[i].term) == 0) {
73                 globalTerms[j].frequency += buffer[i].frequency;
74                 exists = 1;
```

Analyze_segment function: Processes a specific file segment to extract words and update the local buffer.

Integrate_results: Merges results from a local buffer into the global shared memory.

```
75                 break;
76             }
77         }
78         // Add new term if not found
79         if (!exists) {
80             if (*globalTermCount < MAX_TERMS) {
81                 strcpy(globalTerms[*globalTermCount].term, buffer[i].term);
82                 globalTerms[*globalTermCount].frequency = buffer[i].frequency;
83                 (*globalTermCount)++;
84             } else {
85                 fprintf(stderr, "Shared memory limit exceeded!\n");
86             }
87         }
88     }
89 }
90
91 // Identify and print the top 10 most frequent terms
92 void top_10_terms() {
93     // Sort terms by frequency in descending order
94     for (int i = 0; i < 10; i++) {
95         for (int j = i + 1; j < *globalTermCount; j++) {
96             if (globalTerms[j].frequency > globalTerms[i].frequency) {
97                 TermFrequency temp = globalTerms[i];
98                 globalTerms[i] = globalTerms[j];
99                 globalTerms[j] = temp;
100            }
101        }
102    }
103
104    printf("Top 10 terms:\n");
105    // Print the top 10 terms and their frequencies
106    for (int i = 0; i < 10 && i < *globalTermCount; i++) {
107        printf("%s: %d\n", globalTerms[i].term, globalTerms[i].frequency);
108    }
109 }
110
111 int main() {
```

top_10_terms: Sorts and prints the top 10 terms by frequency.

```c
int main() {
    const char *filename = "text8.txt";
    FILE *inputFile = fopen(filename, "r");
    if (!inputFile) {
        perror("Failed to open file");
        return 1;
    }

    // Get the total size of the file
    fseek(inputFile, 0, SEEK_END);
    int totalSize = ftell(inputFile);
    int chunkSize = totalSize / PROCESS_COUNT;  // Divide file into equal chunks for each process
    fclose(inputFile);

    // Map shared memory for global term frequencies and count
    globalTerms = mmap(NULL, MAX_TERMS * sizeof(TermFrequency), PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1, 0);
    globalTermCount = mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1, 0);
    *globalTermCount = 0;  // Initialize the term count to 0

    struct timespec begin, finish;
    clock_gettime(CLOCK_MONOTONIC, &begin);  // Start timer

    // Fork child processes to handle different file segments
    for (int id = 0; id < PROCESS_COUNT; id++) {
        int pid = fork();
        if (pid == 0) {
            TermFrequency *localBuffer = malloc(LOCAL_STORAGE * sizeof(TermFrequency));
            int bufferCount = 0;

            int start_offset = id * chunkSize;
            int end_offset = (id == PROCESS_COUNT - 1) ? totalSize : (id + 1) * chunkSize;
            analyze_segment(filename, start_offset, end_offset, localBuffer, &bufferCount);

            // Write the local results to a temporary file
            char tempFile[20];
            sprintf(tempFile, "segment_%d.bin", id);
            FILE *temp = fopen(tempFile, "wb");
```

File Setup: file size and determines the chunk size for each process.

Shared Memory Setup: Allocates shared memory for storing terms and their frequencies.

Child Processes: Creates child processes to process file segments and save results to temporary files.

```c
            fwrite(&bufferCount, sizeof(int), 1, temp);
            fwrite(localBuffer, sizeof(TermFrequency), bufferCount, temp);
            fclose(temp);

            free(localBuffer);
            exit(0);
        }
    }

    // Wait for all child processes to finish
    for (int id = 0; id < PROCESS_COUNT; id++) {
        wait(NULL);
    }

    // Consolidate results from all child processes
    for (int id = 0; id < PROCESS_COUNT; id++) {
        char tempFile[20];
        sprintf(tempFile, "segment_%d.bin", id);
        FILE *temp = fopen(tempFile, "rb");

        int bufferCount;
        fread(&bufferCount, sizeof(int), 1, temp);
        TermFrequency *localBuffer = malloc(bufferCount * sizeof(TermFrequency));
        fread(localBuffer, sizeof(TermFrequency), bufferCount, temp);

        integrate_results(localBuffer, bufferCount);

        free(localBuffer);
        fclose(temp);
        remove(tempFile);  // Clean up temporary files
    }

    // Print the top 10 most frequent terms
    top_10_terms();

    clock_gettime(CLOCK_MONOTONIC, &finish);  // End timer
    double duration = (finish.tv_sec - begin.tv_sec) + (finish.tv_nsec - begin.tv_nsec) / 1e9;
```

```
    clock_gettime(CLOCK_MONOTONIC, &finish);  // End timer
    double duration = (finish.tv_sec - begin.tv_sec) + (finish.tv_nsec - begin.tv_nsec) / 1e9;
    printf("Execution time: %f seconds\n", duration);

    // Unmap shared memory
    munmap(globalTerms, MAX_TERMS * sizeof(TermFrequency));
    munmap(globalTermCount, sizeof(int));

    return 0;
}
```

Figure 6: code for multiprocessing

Parent Process Consolidation: Merges results from temporary files into global shared memory.

Execution Time and Cleanup: Measures and prints execution time. Cleans up shared memory.

Results for using 2 processes:

Figure 7: results for 2 processes

Results for using 4 processes:

Figure 8: results for 4 processes

Results for using 6 processes:

```
basmala@basmala-VirtualBox:~$ ./proj
Top 10 terms:
the: 1061396
of: 593677
and: 416629
one: 411764
in: 372201
a: 325873
to: 316376
zero: 264975
nine: 250430
two: 192644
Execution time: 169.423821 seconds
basmala@basmala-VirtualBox:~$
```

Figure 9: results for 6 processes

Results for using 6 processes:

```
basmala@basmala-VirtualBox:~$ ./proj
Top 10 terms:
the: 1061396
of: 593677
and: 416629
one: 411764
in: 372201
a: 325873
to: 316376
zero: 264975
nine: 250430
two: 192644
Execution time: 173.765683 seconds
basmala@basmala-VirtualBox:~$
```

Figure 10: results for 8 processes

Discussion:

- **Creating shared memoey:**

In this approach, shared memory is used to allow multiple child processes to communicate and store their results. Instead of pipes, the program utilizes `mmap` to allocate a shared memory region for storing global term frequencies and a counter for the total number of terms. The `globalTerms` pointer points to this shared memory, where each child process will store the term frequencies it computes for its portion of the dataset. Similarly, `globalTermCount` is used to keep track of the number of unique terms across all segments. Shared memory facilitates efficient inter-process communication by allowing all processes to access and modify the same memory region.

- **Creating child processes:**

The fork() system call is used to create multiple child processes in the code. Each child process is assigned a specific segment of the file to process. The fork() function returns 0 in the child process and a positive process ID in the parent process. The child processes independently analyze their assigned file segments, computing term frequencies and storing them in their local buffer. Once all child processes have completed their computations, the parent process waits for all child processes to finish using the wait() system call. Afterward, the parent process consolidates the results stored in the shared memory to produce the final term frequencies.

- **Multiprocessing for listing top 10 most frequented words:**

This C program processes a large text file to count the frequency of each unique term and identifies the top 10 most frequent terms. It uses multiple processes for parallel execution, dividing the file into chunks and processing each chunk independently to speed up the computation. The results from each process are stored in local memory and then consolidated into shared memory in the parent process. Shared memory is used to store the global list of terms and their frequencies, ensuring that all processes can update this data. Each process writes its local results to a temporary file, which is later read by the parent process to integrate the findings. The program measures and prints the total execution time. The sorting of terms is done in the parent process to identify the top 10 terms. The program uses `mmap` for shared memory management and `fork` for process creation, and temporary files are used to handle large data between processes. This approach leverages parallel processing and shared memory to efficiently process large datasets in a concurrent environment.

- **Performance Evaluation and Analysis:**

The performance of the multiprocessing approach can be evaluated by measuring the execution time using `clock_gettime()` or other timing methods. The time taken for each child process to analyze its segment of the file and for the parent to aggregate the results can be compared against a sequential approach to measure the speedup achieved by parallel processing. By varying the number of child processes, the efficiency and scalability of the solution can be assessed. The number of child processes should ideally be chosen based on the number of available CPU cores to maximize performance.

- Performance (Throughput) = 1/time

➢ **By Trying Different Number of processes:**

Number of processes	**Execution Time**	**Performance**
2 processes	254.862480	0.00392368
4 processes	205.416934	0.00486815
6 processes	169.423821	0.00590236
8 processes	173.765683	0.00575488

➢ **Impact of Process Count**

The quantity of processes directly shapes execution time and overall efficiency. Increasing the number of processes typically accelerates task completion and boosts throughput by harnessing parallelism. Nevertheless, an abundance of processes can introduce overhead, vying for resources and potentially dampening productivity. Through experimentation on a system equipped with 7 CPUs, employing 6 processes emerged as the optimal strategy, aligning with hardware constraints to mitigate overhead and optimize performance.

3. Multithreading approach:

Multithreading allows concurrent execution within a single process, with threads sharing the same memory space. Tasks are broken down into smaller subtasks, each handled by a separate thread within the same process. Direct communication through shared memory enables more efficient synchronization compared to inter-process communication. This approach enhances performance for tasks with high parallelism and shared memory requirements. However, it demands careful synchronization to prevent issues like data races and deadlocks. Despite these challenges, multithreading generally incurs lower overhead than multiprocessing, making it ideal for parallelizing tasks with fine-grained parallelism.

Code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <ctype.h>
#include <time.h>

#define WORD_LEN 50          // Maximum length of a word
#define NUM_THREADS 8        // Number of threads to use
#define MAX_WORDS 17500000   // Maximum number of words that can be stored

// Structure to store a word and its count
typedef struct {
    char word[WORD_LEN];
    int count;
} WordCount;

// Global data structure for storing words and their counts
WordCount *global_words;
int global_word_count = 0;
pthread_mutex_t lock;        // Mutex for synchronizing access to the global structure

// Thread-local storage structure
typedef struct {
    WordCount *local_words;  // Array to store words and counts for each thread
    int local_word_count;    // Number of words stored in the thread-local array
} ThreadData;

// Structure to pass arguments to threads
typedef struct {
    int thread_id;           // ID of the thread
    int segment_start;       // Start position of the file segment assigned to the thread
    int segment_end;         // End position of the file segment assigned to the thread
    ThreadData *thread_data; // Pointer to thread-local data
} ThreadArgs;

// Find the index of a word in the array of words
```

here, I defined a struct for words and its count, and a struct for threadData contains local_words: An array of WordCount structures for thread-local word counts. local_word_count: The number of unique words stored locally. Finally a struct for threadsArgs, Arguments passed to threads: thread_id: The ID of the thread. segment_start and segment_end: File segment boundaries the thread is responsible for.
thread_data: A pointer to the thread's ThreadData.

```c
int find_word_index(WordCount *words, int word_count, const char *word) {
    for (int i = 0; i < word_count; i++) {
        if (strcmp(words[i].word, word) == 0) {
            return i; // Word found, return its index
        }
    }
    return -1; // Word not found
}

// Add a word to the thread-local storage
void add_word(ThreadData *data, const char *word) {
    int index = find_word_index(data->local_words, data->local_word_count, word);
    if (index >= 0) {
        data->local_words[index].count++; // Increment count if word already exists
        return;
    }
    if (data->local_word_count >= MAX_WORDS / NUM_THREADS) {
        fprintf(stderr, "Thread-local storage exceeded\n");
        exit(1); // Exit if storage limit is exceeded
    }
    strcpy(data->local_words[data->local_word_count].word, word); // Add new word
    data->local_words[data->local_word_count].count = 1;          // Initialize count to 1
    data->local_word_count++;
}

// Process a file segment assigned to the thread
void *process_chunk(void *arg) {
    ThreadArgs *args = (ThreadArgs *)arg;

    // Open the file
    FILE *file = fopen("text8.txt", "r");
    if (!file) {
        perror("Failed to open file");
        pthread_exit(NULL);
    }

    // Move to the start of the segment
```

Find_word_index function: Searches for a word in the provided array of WordCount.

Add_word function: Adds a word to the thread's local storage.

Process_chunk function: Processes a specific file segment to extract words.

```c
    fseek(file, args->segment_start, SEEK_SET);

    if (args->segment_start > 0) {
        // Ensure the start position is at a word boundary
        char c;
        while (!feof(file)) {
            c = fgetc(file);
            if (isspace(c)) break;
        }
    }

    char word[WORD_LEN];
    // Read words within the segment
    while (ftell(file) < args->segment_end && fscanf(file, "%49s", word) != EOF) {
        add_word(args->thread_data, word); // Add word to thread-local storage
    }

    fclose(file);
    return NULL;
}

// Merge thread-local results into the global array
void merge_results(ThreadData *thread_data) {
    pthread_mutex_lock(&lock); // Lock mutex for synchronization
    for (int i = 0; i < thread_data->local_word_count; i++) {
        int index = find_word_index(global_words, global_word_count, thread_data->local_words[i].word);
        if (index >= 0) {
            global_words[index].count += thread_data->local_words[i].count; // Update count if word exists
        } else {
            if (global_word_count >= MAX_WORDS) {
                fprintf(stderr, "Global word storage exceeded\n");
                pthread_mutex_unlock(&lock);
                exit(1); // Exit if global storage limit is exceeded
            }
            // Add new word to global storage
            strcpy(global_words[global_word_count].word, thread_data->local_words[i].word);
            global_words[global_word_count].count = thread_data->local_words[i].count;
```

Merge_results function: Merges a thread's local results into the global word array. Uses a mutex to synchronize access to global storage. Prevents race conditions with pthread_mutex_lock.

```
            global_word_count++;
        }
    }
    pthread_mutex_unlock(&lock); // Unlock mutex
}

// Find and print the top 10 most frequent words
void find_top_10_words() {
    // Sort the global array by word count in descending order
    for (int i = 0; i < global_word_count - 1; i++) {
        for (int j = i + 1; j < global_word_count; j++) {
            if (global_words[j].count > global_words[i].count) {
                WordCount temp = global_words[i];
                global_words[i] = global_words[j];
                global_words[j] = temp;
            }
        }
    }

    // Print the top 10 words
    printf("Top 10 words:\n");
    for (int i = 0; i < 10 && i < global_word_count; i++) {
        printf("%s: %d\n", global_words[i].word, global_words[i].count);
    }
}

int main() {
    // Open the file and determine its size
    FILE *file = fopen("text8.txt", "r");
    if (!file) {
        perror("Failed to open file");
        return 1;
    }

    fseek(file, 0, SEEK_END);
    int file_size = ftell(file);
    fclose(file);
```

Find_top_10_words function: Sorts and prints the top 10 most frequent words.

```
    // Allocate memory for the global word array
    global_words = malloc(MAX_WORDS * sizeof(WordCount));
    if (!global_words) {
        perror("Failed to allocate global memory");
        return 1;
    }

    pthread_mutex_init(&lock, NULL); // Initialize mutex

    pthread_t threads[NUM_THREADS];
    ThreadArgs thread_args[NUM_THREADS];
    ThreadData thread_data[NUM_THREADS];
    int segment_size = file_size / NUM_THREADS; // Calculate segment size for each thread

    struct timespec start, end;
    clock_gettime(CLOCK_MONOTONIC, &start); // Start time measurement

    // Create threads and assign file segments
    for (int i = 0; i < NUM_THREADS; i++) {
        thread_data[i].local_words = malloc((MAX_WORDS / NUM_THREADS) * sizeof(WordCount));
        thread_data[i].local_word_count = 0;

        thread_args[i].thread_id = i;
        thread_args[i].segment_start = i * segment_size;
        thread_args[i].segment_end = (i == NUM_THREADS - 1) ? file_size : (i + 1) * segment_size;
        thread_args[i].thread_data = &thread_data[i];

        pthread_create(&threads[i], NULL, process_chunk, &thread_args[i]);
    }

    // Wait for threads to complete and merge their results
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
        merge_results(&thread_data[i]);
        free(thread_data[i].local_words);
    }

    find_top_10_words(); // Find and print the top 10 words

    clock_gettime(CLOCK_MONOTONIC, &end); // End time measurement
    double elapsed_time = (end.tv_sec - start.tv_sec) + (end.tv_nsec - start.tv_nsec) / 1e9;
    printf("Execution time (multithreading): %.6f seconds\n", elapsed_time);

    pthread_mutex_destroy(&lock); // Destroy mutex
    free(global_words);           // Free allocated memory

    return 0;
}
```

Figure 11: multithreading code

Main function:

- Calculates file size and segment size.
- Creates threads to process segments.
- Merges thread-local results into global storage.
- Finds and prints the top 10 words.
- Measures and prints execution time.
- Frees memory and cleans up resources.

Results for using 2 threads:

```
basmala@basmala-VirtualBox:~$ ./proth
Top 10 words:
the: 1061396
of: 593677
and: 416629
one: 411764
in: 372201
a: 325873
to: 316376
zero: 264975
nine: 250430
two: 192644
Execution time (multithreading) : 549.900396 seconds
basmala@basmala-VirtualBox:~$
```

Figure 12: results for 2 threads

Results for using 4 threads:

```
basmala@basmala-VirtualBox:~$ ./proth
Top 10 words:
the: 1061396
of: 593677
and: 416629
one: 411764
in: 372201
a: 325873
to: 316376
zero: 264975
nine: 250430
two: 192644
Execution time (multithreading) : 227.766020 seconds
basmala@basmala-VirtualBox:~$
```

Figure 13: results for 4 threads

Results for using 6 threads:

```
basmala@basmala-VirtualBox:~$ ./proth
Top 10 words:
the: 1061396
of: 593677
and: 416629
one: 411764
in: 372201
a: 325873
to: 316376
zero: 264975
nine: 250430
two: 192644
Execution time (multithreading) : 238.431504 seconds
basmala@basmala-VirtualBox:~$
```

Figure 14: results for 6 threads

Results for using 8 threads:

```
basmala@basmala-VirtualBox:~$ ./proth
Top 10 words:
the: 1061396
of: 593677
and: 416629
one: 411764
in: 372201
a: 325873
to: 316376
zero: 264975
nine: 250430
two: 192644
Execution time (multithreading) : 283.869193 seconds
basmala@basmala-VirtualBox:~$
```

Figure 15: results for 8 threads

Discussion:

- **Shared Memory for Global Results**:

  In the multithreading approach, a shared memory (`global_words` array) is used for storing the global term frequencies and the total count of unique words. Synchronization is achieved through a `pthread_mutex_t` lock to prevent data races when threads update the global word array. This shared-memory model allows efficient inter-thread communication since threads within the same process can directly access shared data without the need for additional inter-process communication mechanisms like pipes or files.

- **Thread Creation**:

  The program creates multiple threads using `pthread_create`, each responsible for processing a specific segment of the file. Threads independently analyze their assigned file segments, compute term frequencies, and store results in thread-local memory. After processing, results from each thread are merged into the shared global memory using the `merge_results` function.

- **Multithreading for Top 10 Most Frequent Words**:

  The multithreaded design processes a large text file to count term frequencies and identify the top 10 most frequent terms. Threads work in parallel to process distinct segments of the file, and results are consolidated by the main thread. Sorting to find the top 10 terms is performed by the main thread after merging results.

- **Performance Evaluation**:

  Performance can be measured using `clock_gettime()` to record the start and end time of the computation. The execution time for the multithreaded approach is expected to be significantly lower than a sequential approach due to parallel processing.

- **Scalability**:

  The scalability of the multithreading approach depends on the number of threads (`NUM_THREADS`) and the number of available CPU cores. Overloading threads beyond the number of available cores can lead to context-switching overhead, diminishing returns on performance.

- **Efficiency**:

  The efficiency of the solution can be evaluated by comparing the speedup achieved using multiple threads versus a single-threaded sequential approach.

  - Performance (Throughput) = 1/time

➢ **By Trying Different Number of processes:**

Number of threads	**Execution Time**	**Performance**
2 threads	549.900396	0.00181851
4 threads	227.766020	0.00439047
6 threads	238.431504	0.00419407
8 threads	283.869193	0.00352246

➢ **Impact of thread Count:**

The impact of thread count reveals that performance improves as the number of threads increases up to an optimal level, aligning with the number of available CPU cores. In this case, using 4 threads achieves the best performance with the lowest execution time (227.77 ms) and highest throughput (0.00439). Beyond this, performance diminishes due to oversubscription of CPU cores, leading to increased context-switching overhead. For instance, at 6 and 8 threads, execution time increases to 238.43 ms and 283.87 ms, respectively, while throughput drops. This indicates that excessive threading causes contention and synchronization overhead, highlighting the importance of optimizing thread count based on system resources.

Performance Measurements:

Number	Execution Time	Performance
Naïve (Sequential)	619.288107	0.00161476
2 processes	254.862480	0.00392368
4 processes	205.416934	0.00486815
6 processes	169.423821	0.00590236
8 processes	173.765683	0.00575488
2 threads	549.900396	0.00181851
4 threads	227.766020	0.00439047
6 threads	238.431504	0.00419407
8 threads	283.869193	0.00352246

- **Naïve (Sequential)**:

  Performance: The slowest with an execution time of **619.28 ms** and throughput of **0.00161**. It processes the entire dataset serially, using only one CPU core and not leveraging parallelism.

  Overhead: Minimal as there is no inter-process or inter-thread communication, but its lack of concurrency limits performance.

- **Multiprocessing**:

  Best Execution Time: Achieved with **6 processes** (169.42 ms) and throughput of **0.00590**.

  Performance Trend: Increases with the number of processes up to 6 but slightly decreases with 8 due to overhead. Because Processes run independently and fully utilize multiple CPU cores. Shared memory (`mmap`) efficiently aggregates results.

  Overhead: Context-switching and shared-memory synchronization become significant with 8 processes, leading to diminished performance.

- **Multithreading**:

  Best Execution Time: Achieved with **4 threads** (227.77 ms) and throughput of **0.00439**.

  Performance Trend: Improves with up to 4 threads but decreases with more threads. Because Threads efficiently share memory within the same process, but oversubscription (e.g., 6 or 8 threads) introduces contention and mutex lock overhead.

  Overhead: Mutex-based synchronization is necessary to avoid data races, which becomes costly when threads exceed available cores.

Which is Best and Why?

- **Best Method**: **6 Processes (Multiprocessing)** achieves the highest throughput of **0.00590** and lowest execution time of **169.42 ms**.
- **Reason**: Multiprocessing fully utilizes CPU cores with isolated memory spaces for each process, reducing synchronization overhead compared to multithreading. Shared memory (mmap) aggregates results efficiently.
- **Overhead in Methods**:

  Naïve: No concurrency overhead, but underutilizes resources.

  Multiprocessing: Overhead from context-switching and shared memory increases with more processes but remains manageable at 6.

  Multithreading: Overhead from mutex locks and context-switching rises significantly with more threads than CPU cores.

Analysis According to Amdahl's Law:

Amdahl's Law defines the theoretical maximum speedup S of a program based on the parallelizable and serial portions of the code:

1. **naive Approach:**

serial part: readWords and listTopWords
time_serial: Measures the time for serial parts (readWords and listTopWords).

Parallel part: processWords
time_parallel: Measures the time for the parallel part (processWords).

after running the code for 5 times I took the avg. which is:
serial part = 1.6783844 sec
parallel part = 608.101722

serial = serial part / parallel part + serial part = 0.2752442 100/100
parallel = 1- serial = 99.724756 100/100

speedup = s = 1 / ((1-p) + p / n) = 13.5263489

maximum speedup = smax = 1 / 1 – p = 363.314 if n goes to infinity

2. **Multiprocessing Approach:**

Serial Section:

The serial execution happens in the following parts:

1. **File Handling**: Opening the file and determining its size using fopen() and fseek() in the main() function is a serial operation. This is performed once at the start of the program to divide the file into segments.
2. **Waiting for Child Processes**: The wait() function ensures that the parent process waits for all child processes to complete their tasks before proceeding. This synchronization is inherently serial.
3. **Integrating Results**: After all child processes complete, the parent process reads and integrates results from the temporary files in the integrate_results() function. This sequentially combines the local term frequencies into the global array.
4. **Sorting the Results**: Sorting the global array to determine the top 10 most frequent terms is a serial operation. This is done in the top_10_terms() function.

Parallel Section:

The parallel execution occurs in the following parts:

1. **Process Creation and Execution**: The fork() function creates child processes that independently analyze different segments of the file. Each child process executes the analyze_segment() function to read its assigned portion of the file and compute term frequencies in parallel.
2. **File Segment Analysis**: Each child process analyzes its assigned segment of the file using fseek() and fscanf() in parallel. The processing of terms and updating the local buffer occurs independently for each process.
3. **Temporary File Writing**: Each child process writes its local results into separate temporary files. This ensures that processes do not interfere with each other while working in parallel.

3. MultiThreading Approach:

Serial Section:

The serial execution happens in the following parts:

1. **File Handling:** Opening the file and determining its size using fopen() and fseek(). This is a serial operation. fopen() is called once, and the file is read sequentially in chunks by the threads. However, the initial file operations are serial.
2. **Sorting the Results:** After the word counts are gathered in the global array, sorting the array to find the top 10 words is done serially in the find_top_10_words() function.
3. **Mutex Locking and Unlocking:** Although it protects shared resources from race conditions, the mutex lock/unlock itself is a serial operation because only one thread can hold the lock at a time.

Parallel Section:

The parallel execution occurs in the following parts:

1. **Thread Creation and Execution:**

   The pthread_create() function creates threads that process different segments of the file in parallel. Each thread works independently to read and process words from its assigned segment of the file. The threads use thread-local storage to store word counts for each segment. The process_chunk() function is executed in parallel by each thread.

2. **Merging Results from Threads:**

After all threads complete their work, the results from each thread (i.e., word counts) are merged into a global word count array using the merge_results() function.

optimal number of child processes or threads:

for multiprocessing = 6 processes

for multithreading = 4 processes

(I explained why in previous pages)

Conclusion

In conclusion, our project delved into the comparative analysis of three distinct parallel computing approaches for calculating average fastest way to read a large data file and listing top 10 most frequented numbers: naïve sequential processing, multiprocessing, and multithreading. the naive multiprocessing approach emerged as the most efficient, boasting the shortest execution time and highest throughput. Among the tested methods, 6 processes achieved the best performance with an execution time of 169.42 ms and throughput of 0.00590, leveraging optimal CPU core utilization and minimal overhead. The multiprocessing approach consistently outperformed multithreading due to the reduced overhead of shared memory synchronization. However, increasing the number of processes beyond 6 led to diminishing returns (e.g., 8 processes slightly increased execution time to 173.77 ms) due to overhead from context switching and shared memory contention. Multithreading, 4 threads provided the best balance between performance and overhead, with an execution time of 227.77 ms and throughput of 0.00439. Using more threads resulted in performance degradation due to mutex lock contention and oversubscription of CPU cores. And faced challenges in achieving optimal performance and exhibited longer execution times compared to multiprocessing. Overall, 6 processes is the most effective configuration for this workload, maximizing the benefits of parallel processing while avoiding the overhead associated with oversubscription and synchronization. These findings underscore the importance of selecting the appropriate parallel computing strategy based on task requirements and system constraints.