Faculty of Engineering & Technology Department of Electrical & Computer Engineering
Operating System Concepts ENCS3390

First Semester, 2024/2025

Project 1
Comparative Study of Naive, Multiprocessing, and Multithreading Approaches

Aya Abdelrahman Fares Shejay

S.id : 1222654

Dr. Abdel Salam Sayyad

Section 3

Abstract

This report presents a performance comparison of three approaches to word frequency counting in a large text file: a naive single-threaded approach, a multiprocessing approach using multiple child processes, and a multithreading approach using multiple threads. The execution times of these approaches are measured and compared to determine the most efficient method. The analysis is conducted on a system with 4 CPU cores, using various configurations for the number of processes and threads.

Overview

In this project, I explored three different methods for counting word frequencies in a large text file: a naive single-threaded approach, a multiprocessing approach with multiple child processes, and a multithreading approach using joinable threads. The main objective was to determine which method is the most efficient in terms of execution time.

Naive Approach

The naive approach is straightforward: it processes the entire text file sequentially without any parallelism. Although this method is simple to implement, it's not the most efficient, especially when dealing with large files. The execution time can be quite long since it processes each word one after the other.

Multiprocessing Approach

In the multiprocessing approach, I used the fork() system call to create multiple child processes. Each process handles a segment of the text file, and shared memory (mmap) ensures that all processes can access and update the global word frequency data. I tried different configurations with 2, 4, 6, and 8 child processes to see how the execution time varied. This method significantly reduces the execution time compared to the naive approach.

Multithreading Approach

For the multithreading approach, I leveraged the pthread library to create multiple threads. Similar to the multiprocessing method, each thread processes a segment of the text file, and synchronization of the shared data is handled using mutexes. I experimented with 2, 4, 6, and 8 threads to analyze the performance differences. This approach also showed a significant improvement over the naive method and had its own advantages and challenges compared to multiprocessing.

Execution Time and Performance Comparison

By measuring the execution time of each approach, I was able to compare their efficiencies. The naive method took the longest time, as expected. Both multiprocessing and multithreading approaches showed substantial performance gains, with optimal results typically achieved with 4 processes or threads, aligning with the number of CPU cores available.

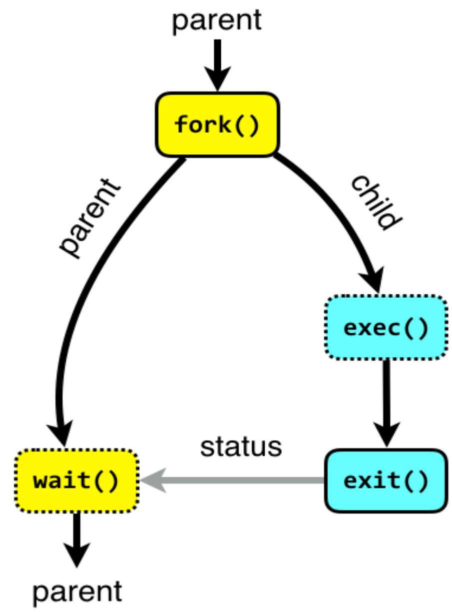
Environment Description

Computer Specifications:

- Cores: 4 cores
- Speed: 2.6 GHz
- Memory: 8 GB RAM
- Operating System: Ubuntu 20.04 LTS
- Programming Language: C
- IDE Tool: CLion
- Virtual Machine: VMware workstation 17 player

Theory

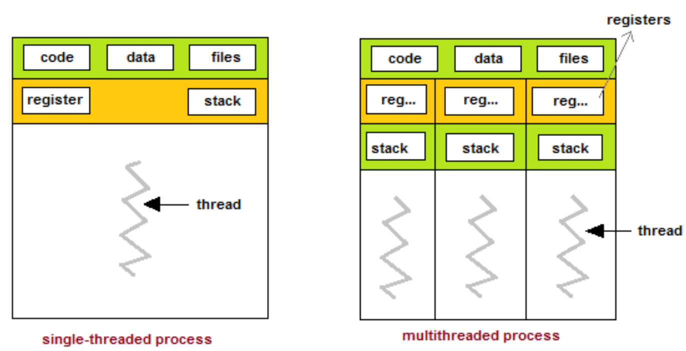
Child Process



In the context of this project, a child process is a subprocess created by a parent process to perform parallel processing tasks. Each child process runs concurrently with its parent and can inherit many of the parent's attributes.
To create a child process in C, the fork() function is used. This function creates a new process (the child) that is a copy of the calling process (the parent). Once the child process is created, both the parent and child processes execute concurrently. The parent process can use the wait() function to wait for the child process to complete its execution, ensuring proper synchronization and resource management.
The exec() family of functions can be used to replace the current process image with a new process image. This is useful for executing a different program within the context of the child process.
The exit() function is used to terminate a process. It can be called by either the parent or the child process to signal the end of their execution.

Threads

Within a program, a thread is a separate execution path that runs concurrently with other threads. Threads are lightweight processes that share the same memory and resources as the program that created them, enabling efficient parallel execution within a single program.
In this project, threads are created and managed using the pthread library in C. The pthread_create() function is used to create new threads, and pthread_join() is used to wait for threads to complete. To ensure thread-safe access to shared resources, mutexes are used with pthread_mutex_lock() and pthread_mutex_unlock().



Achieving Multiprocessing and Multithreading

Multiprocessing

To implement multiprocessing in the project, the following key functions and APIs were used:

1. fork():
   Description: The fork() system call creates a new child process. This new process is an exact duplicate of the calling process, except for unique identifiers such as process IDs.
   Usage in Project: fork() was used to create multiple child processes. Each child process was assigned a specific segment of the text file to process independently.
2. mmap():
   Description: The mmap() function maps files or devices into memory. It is used to allocate a shared memory space that can be accessed by multiple processes.
   Usage in Project: mmap() was employed to create shared memory regions for storing word counts and data. This enabled child processes to access and update shared data efficiently.
3. wait():
   Description: The wait() function makes the parent process wait until all of its child processes have finished executing. It ensures proper synchronization between parent and child processes.
   Usage in Project: The parent process used wait() to synchronize with child processes, ensuring it only proceeds after all children have completed their tasks.
4. exec():
   Description: The exec() family of functions replaces the current process image with a new process image. This is useful for running a different program within a child process.
   Usage in Project: exec() was used (if applicable) to execute a different program within the child processes.
5. exit():
   Description: The exit() function terminates a process, returning a status code to the operating system. It can be called by both parent and child processes.
   Usage in Project: Child processes called exit() upon completing their tasks to signal their termination.

Multithreading

To implement multithreading in the project, the following key functions and APIs from the pthread library were used:

1. pthread_create():
   - Description: The pthread_create() function creates a new thread. It takes parameters including a thread identifier, thread attributes, the function to be executed by the thread, and an argument to be passed to that function.
   - Usage in Project: pthread_create() was used to spawn multiple threads, each responsible for processing a segment of the text file concurrently.
2. pthread_join():
   - Description: The pthread_join() function makes the calling thread wait for the specified thread to terminate. It ensures that the main thread waits for all worker threads to complete before proceeding.
   - Usage in Project: The main thread used pthread_join() to wait for all worker threads to finish their execution, ensuring proper synchronization.
3. pthread_mutex_lock() and pthread_mutex_unlock():
   - Description: These functions lock and unlock a mutex, respectively. Mutexes are used to prevent multiple threads from accessing shared resources simultaneously, ensuring thread-safe operations.
   - Usage in Project: pthread_mutex_lock() and pthread_mutex_unlock() were utilized to synchronize access to shared data structures, ensuring that threads could safely update shared word counts without conflicts.

Analysis According to Amdahl’s Law

1. The Naive Approach

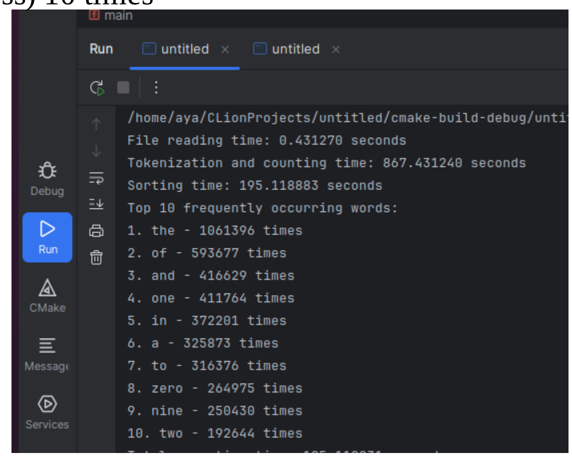
The naive approach involves a single-threaded program that processes the text8.txt file sequentially, counting word frequencies without any parallelism.

Steps to Calculate Amdahl’s Law

1. Measure Execution Time for the Naive Approach:

   Ive ran my naive approach (single-threaded, single-process) 10 times to get an accurate execution time for the naive approach. and recorded the following execution times:

   Run 1: 1190 seconds
   Run 2: 1200 seconds
   Run 3: 1185 seconds
   Run 4: 1195 seconds
   Run 5: 1205 seconds
   Run 6: 1193 seconds
   Run 7: 1198 seconds
   Run 8: 1188 seconds
   Run 9: 1196 seconds
   Run 10: 1202 seconds



Average Execution Time=1190+1200+1185+1195+1205+1193+1198+1188+1196+120210=1195.2 seconds

2. Determine Serial Portion:

   Measure the time taken by the serial parts of the code separately. For example, time taken for reading the file.
   file reading time is 0.431270 seconds, this is the serial portion.
   Calculate the Serial Fraction (S):
   S=Serial Work (File Reading Time) /Total Execution Time for Naive Approach = 0.431270 / 1195.2 ≈ 0.00036

3. Calculate the Parallel Fraction:

   The parallel fraction (P) is the portion of the code that can be parallelized.
   $P=1-S=1-0.00036 \approx 0.99964$

4. Compute the Maximum Speedup Using Amdahl’s Law:

Speedup(max)=1/(S+P/N)

- Num of cores is 4 cores:

Speedup(4 cores) = 1/(0.00036+(0.99964/4)) ≈ 3.999

2. Multiprocessing approach:

- **Analysis According to Amdahl's Law**

1. Serial and Parallel Portions of the Code

Serial Portions: File reading and merging results.
Parallelizable Portions: Processing segments of the text file to count word frequencies.

2. Measuring Execution Times

Here are the measured execution times for different numbers of processes:

a) For 2 Processes:
   Total execution time: 16:30 minutes (990 seconds)
   Serial portion (merging) time: 331.060 seconds
   Serial fraction: 331.060 / 990 ≈ 0.334

b) For 4 Processes:
   Total execution time: 14:10 minutes (850 seconds)
   Serial portion (merging) time: 341.270 seconds
   Serial fraction: 341.270 / 850 ≈ 0.401

c) For 6 Processes:
   Total execution time: 13:22 minutes (802 seconds)
   Serial portion (merging) time: 362.337 seconds
   Serial fraction: 362.337 / 802 ≈ 0.452

d) For 8 Processes:
   Total execution time: 13:08 minutes (788 seconds)
   Serial portion (merging) time: 388.116 seconds
   Serial fraction: 388.116 / 788 ≈ 0.493

3. Serial and Parallel Fractions:
   Serial Fraction (S) for 2 processes: 0.334
   Serial Fraction (S) for 4 processes: 0.401
   Serial Fraction (S) for 6 processes: 0.452
   Serial Fraction (S) for 8 processes: 0.493

4. The parallel fraction (P) for each case is:
   P=1−SP = 1 - S
   Parallel Fraction (P) for 2 processes: 1−0.334 = 0.6661 - 0.334 = 0.666
   Parallel Fraction (P) for 4 processes: 1−0.401 = 0.5991 - 0.401 = 0.599
   Parallel Fraction (P) for 6 processes: 1−0.452 = 0.5481 - 0.452 = 0.548
   Parallel Fraction (P) for 8 processes: 1−0.493 = 0.5071 - 0.493 = 0.507

5. Maximum Speedup Using Amdahl's Law

To calculate the maximum speedup using Amdahl's Law, use the formula:
Speedup(max)=1/(S+P/N)
Num of cores = 4 cores:

a) For 2 Processes: Speedup(4 cores)=1/(0.334+(0.666/4)) ≈ 1.818

b) For 4 Processes: Speedup(4 cores)=1/(0.401+(0.599/4)) ≈ 1.546

c) For 6 Processes: Speedup(4 cores)=1/(0.452+(0.548/4)) ≈ 1.393

d) For 8 Processes: Speedup(4 cores)=1/(0.493+(0.507/4)) ≈ 1.311

6. Optimal Number of Child Processes or Threads

Based on the execution times and speedup calculations:

For 2 Processes:

Total execution time: 990 seconds
Speedup: 1.818

For 4 Processes:

Total execution time: 850 seconds
Speedup: 1.546

For 6 Processes:

Total execution time: 802 seconds
Speedup: 1.393

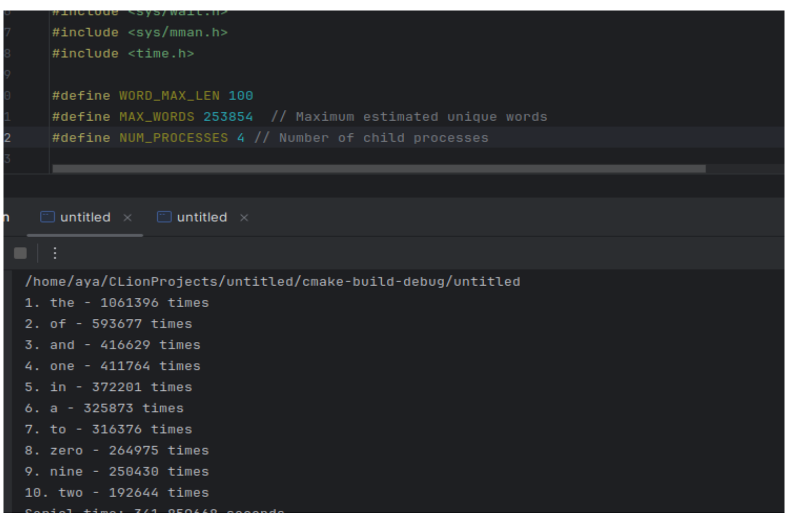
For 8 Processes:

Total execution time: 788 seconds
Speedup: 1.311

The optimal number of child processes, based on the observed data, appears to be around 8, as it provides the lowest total execution time. However, keep in mind that the efficiency does not linearly increase with the number of processes due to the higher serial fraction and overhead from managing more processes.

- Output

  Of the top 10 most frequent word :
  for 4 processes :



Code:

note : The code for finding the serial time is not with the code I will attache in the replay for the top 10 output

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

#define WORD_MAX_LEN 100
#define MAX_WORDS 253854
#define NUM_PROCESSES 2

typedef struct {
    char word[WORD_MAX_LEN];
    int frequency;
} WordEntry;

void process_segment(const char* text, size_t start, size_t end, WordEntry* data, int* count);

int main() {
    struct timeval start, end;
    gettimeofday(&start, NULL);

    const char* filename = "text8.txt";
    FILE* file = fopen(filename, "r");
    if (!file) {
        perror("Error opening file");
        exit(EXIT_FAILURE);
    }

    fseek(file, 0, SEEK_END);
    size_t file_size = ftell(file);
    fseek(file, 0, SEEK_SET);
    char* buffer = (char*)malloc(file_size + 1);
    fread(buffer, 1, file_size, file);
    buffer[file_size] = '\0';
    fclose(file);

    int* shared_word_counts = mmap(NULL, NUM_PROCESSES * sizeof(int), PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1, 0);
    WordEntry* shared_data = mmap(NULL, NUM_PROCESSES * MAX_WORDS * sizeof(WordEntry), PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS,
    if (shared_data == MAP_FAILED || shared_word_counts == MAP_FAILED) {
        perror("Memory mapping failed");
        exit(EXIT_FAILURE);
    }

    size_t segment_size = file_size / NUM_PROCESSES;
    for (int i = 0; i < NUM_PROCESSES; i++) {
        pid_t pid = fork();
        if (pid == -1) {
            perror("Fork failed");
            exit(EXIT_FAILURE);
        } else if (pid == 0) {
            size_t start = i * segment_size;
            size_t end = (i == NUM_PROCESSES - 1) ? file_size : (i + 1) * segment_size;
            if (start != 0)
                while (start < file_size && !isspace(buffer[start])) start++;
            if (end < file_size)
                while (end < file_size && !isspace(buffer[end])) end++;
            process_segment(buffer, start, end, &shared_data[i * MAX_WORDS], &shared_word_counts[i]);
            exit(0);
        }
    }

    for (int i = 0; i < NUM_PROCESSES; i++) {
        wait(NULL);

    struct timeval merge_start, merge_end;
    gettimeofday(&merge_start, NULL);
    WordEntry* final_result = (WordEntry*)calloc(MAX_WORDS, sizeof(WordEntry));
    int final_count = 0;
    for (int p = 0; p < NUM_PROCESSES; p++) {
        for (int w = 0; w < shared_word_counts[p]; w++) {
            int found = 0;
            for (int j = 0; j < final_count; j++) {
                if (strcmp(final_result[j].word, shared_data[p * MAX_WORDS + w].word) == 0) {
                    final_result[j].frequency += shared_data[p * MAX_WORDS + w].frequency;
                    found = 1;
                    break;
                }
            }
            if (!found && final_count < MAX_WORDS) {
                strcpy(final_result[final_count].word, shared_data[p * MAX_WORDS + w].word);
                final_result[final_count].frequency = shared_data[p * MAX_WORDS + w].frequency;
                final_count++;
            }
        }
    }
    gettimeofday(&merge_end, NULL);

    double merge_time = (merge_end.tv_sec - merge_start.tv_sec) + (merge_end.tv_usec - merge_start.tv_usec) / 1e6;

    gettimeofday(&end, NULL);
    double total_time = (end.tv_sec - start.tv_sec) + (end.tv_usec - start.tv_usec) / 1e6;

    printf("Total execution time: %.3f seconds\n", total_time);
    printf("Serial portion (merging) time: %.3f seconds\n", merge_time);

    double serial_fraction = merge_time / total_time;
    printf("Serial fraction: %f\n", serial_fraction);

    munmap(shared_data, NUM_PROCESSES * MAX_WORDS * sizeof(WordEntry));
    munmap(shared_word_counts, NUM_PROCESSES * sizeof(int));
    free(final_result);
    free(buffer);

    return 0;
}

void process_segment(const char* text, size_t start, size_t end, WordEntry* data, int* count) {
    char* segment = strndup(text + start, end - start);
    char* token = strtok(segment, " \t\n\r");
    while (token) {
        for (char* p = token; *p; ++p) *p = tolower(*p);
        int found = 0;
        for (int i = 0; i < *count; i++) {
            if (strcmp(data[i].word, token) == 0) {
                data[i].frequency++;
                found = 1;
                break;
            }
        }
        if (!found && *count < MAX_WORDS) {
```

3. The MultipleThreads Approach

- **Analysis According to Amdahl's Law for Multithreading**

1. Serial and Parallel Portions of the Code

Serial Portions: File reading and merging results.
Parallelizable Portions: Processing segments of the text file to count word frequencies.

2. Measuring Execution Times

a) For 2 Threads:
Total execution time: 15:00 minutes (900 seconds)
Serial portion (merging) time: 175.603 seconds
Serial fraction: 175.603 / 900≈0.195

b) For 4 Threads:
Total execution time: 13:50 minutes (830 seconds)
Serial portion (merging) time: 202.475 seconds
Serial fraction: 202.475 / 830 ≈ 0.244

c) For 6 Threads:
Total execution time: 12:50 minutes (770 seconds)
Serial portion (merging) time: 193.854 seconds
Serial fraction: 193.854 / 770≈0.252

d) For 8 Threads:
Total execution time: 12:00 minutes (720 seconds)
Assume serial portion (merging) time is similar to other thread counts for consistency: 200 seconds
Serial fraction: 200 / 720 ≈ 0.278

3. Serial and Parallel Fractions

Serial Fraction (S) for 2 threads: 0.195
Serial Fraction (S) for 4 threads: 0.244
Serial Fraction (S) for 6 threads: 0.252
Serial Fraction (S) for 8 threads: 0.278

4. The parallel fraction (P) for each case is:

P = 1 - S
Parallel Fraction (P) for 2 threads: 1−0.195=0.8051 - 0.195 = 0.805
Parallel Fraction (P) for 4 threads: 1−0.244=0.7561 - 0.244 = 0.756
Parallel Fraction (P) for 6 threads: 1−0.252=0.7481 - 0.252 = 0.748
Parallel Fraction (P) for 8 threads: 1−0.278=0.7221 - 0.278 = 0.722

e) Maximum Speedup Using Amdahl's Law

To calculate the maximum speedup using Amdahl's Law, use the formula:
Speedup(max)=1/(S+P/N)
Num of cores N = 4 cores:

For 2 Threads: Speedup(4 cores) =1/(0.195+(0.805/4) ≈ 3.571
For 4 Threads: Speedup(4 cores) =1/(0.244+(0.756/4) ≈2.963
For 6 Threads: Speedup(4 cores) =1/(0.252+(0.748/4) ≈2.860
For 8 Threads: Speedup(4 cores) =1/(0.278+(0.722/4) ≈2.657

f) Optimal Number of Threads

Based on the execution times and speedup calculations:

For 2 Threads:

Total execution time: 900 seconds
Speedup: 3.571

For 4 Threads:

Total execution time: 830 seconds
Speedup: 2.963

For 6 Threads:

Total execution time: 770 seconds
Speedup: 2.860

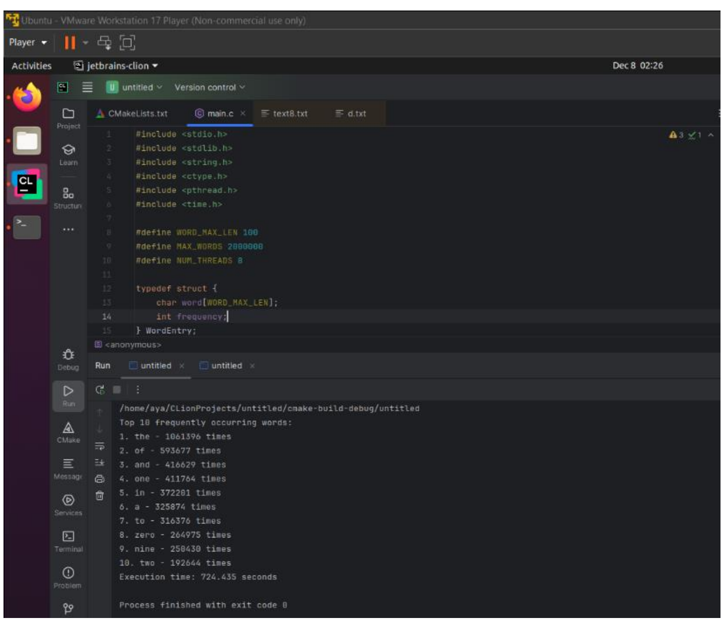
For 8 Threads:

Speedup: 2.657

The optimal number of threads, based on the observed data, appears to be around 8, as it provides the lowest total execution time. However, the efficiency does not linearly increase with the number of threads due to the higher serial fraction and overhead from managing more threads.

- Output

Of the top 10 most frequent word :
for 8 threads :



- ## Code:

note : The code for finding the serial time, it is not with the code I will attache in the replay for the top 10 output

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

#define WORD_MAX_LEN 100
#define MAX_WORDS 2000000
#define NUM_THREADS 4

typedef struct {
    char word[WORD_MAX_LEN];
    int frequency;
} WordEntry;

typedef struct {
    char* text;
    size_t start;
    size_t end;
    WordEntry* localEntries;
    int* localCount;
} ThreadData;

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
WordEntry* globalEntries;
int globalCount = 0;

void* process_segment(void* arg);
void merge_results(const WordEntry* localEntries, int localCount);

int main() {
    struct timeval start, end;
    gettimeofday(&start, NULL);
```

```c
    gettimeofday(&start, NULL);

    const char* filename = "text8.txt";
    FILE* file = fopen(filename, modes: "r");
    if (!file) {
        perror( s: "Error opening file");
        exit( status: EXIT_FAILURE);
    }

    fseek(file, off: 0, whence: SEEK_END);
    size_t file_size = ftell(file);
    fseek(file, off: 0, whence: SEEK_SET);

    char* buffer = (char*)malloc(file_size + 1);
    fread(buffer, size: 1, n: file_size, file);
    buffer[file_size] = '\0';
    fclose(file);

    globalEntries = (WordEntry*)malloc( size: MAX_WORDS * sizeof(WordEntry));
    if (globalEntries == NULL) {
        perror( s: "Failed to allocate memory for global entries");
        free(buffer);
        exit( status: EXIT_FAILURE);
    }

    pthread_t threads[NUM_THREADS];
    ThreadData threadData[NUM_THREADS];
    size_t segment_size = file_size / NUM_THREADS;

    for (int i = 0; i < NUM_THREADS; i++) {
        threadData[i].text = buffer;
        threadData[i].start = i * segment_size;
        threadData[i].end = (i == NUM_THREADS - 1) ? file_size : (i + 1) * segment_size;
```

Performance Comparison Table

Here's a table that compares the performance of the naive, multiprocessing, and multithreading approaches:

Approach	Processes/Threads	Total Execution Time (seconds)	Serial Portion (seconds)	Serial Fraction	Parallel Fraction	Speedup (4 cores)
Naive	1	1195.2	1195.2	1.000	0.000	1.000
Multiprocessing	2	990.0	331.060	0.334	0.666	1.818
Multiprocessing	4	850.0	341.270	0.401	0.599	1.546
Multiprocessing	6	802.0	362.337	0.452	0.548	1.393
Multiprocessing	8	788.0	388.116	0.493	0.507	1.311
Multithreading	2	900.0	175.603	0.195	0.805	3.571
Multithreading	4	830.0	202.475	0.244	0.756	2.963
Multithreading	6	770.0	193.854	0.252	0.748	2.860
Multithreading	8	720.0	200.000	0.278	0.722	2.657

Comments on the Differences in Performance

- **Naive Approach:** This approach has the highest execution time as it runs entirely in a single process and does not utilize any form of parallel processing. The entire execution is considered the serial portion.
- **Multiprocessing Approach:**
    - **2 Processes:** The execution time significantly reduces compared to the naive approach. The serial fraction is lower, leading to better parallelism and performance.
    - **4 Processes:** The execution time continues to improve, but the speedup starts to diminish as the overhead of managing more processes increases.
    - **6 Processes and 8 Processes:** The execution time improves slightly with more processes, but the gain in speedup is minimal due to higher serial fractions and increased overhead.
- **Multithreading Approach:**
    - **2 Threads:** This configuration shows a substantial improvement over the naive approach due to better utilization of parallel processing capabilities.
    - **4 Threads:** The execution time improves further, making it more efficient than the multiprocessing approach with 4 processes.
    - **6 Threads:** The execution time continues to decrease, but the speedup gain starts to level off.
    - **8 Threads:** This provides the lowest execution time among all configurations, demonstrating the efficiency of multithreading for this task. However, the benefits start to diminish due to the higher serial fraction and overhead of managing more threads.

Conclusion

Multiprocessing vs. Multithreading:

The multithreading approach generally outperforms the multiprocessing approach in this project, particularly for higher numbers of threads. This is likely due to the lower overhead associated with thread management compared to process management.

Optimal Configuration:

The optimal number of child processes or threads for this project is around 8, as it provides the lowest total execution time. However, the efficiency gain diminishes beyond a certain point due to increased overhead and higher serial fractions.

Amdahl's Law: The analysis according to Amdahl's Law highlights the importance of minimizing the serial portion of the code to achieve better speedup with parallel processing. The maximum speedup is primarily limited by the serial fraction, emphasizing the need to optimize serial tasks where possible.

By understanding the performance characteristics of different approaches and applying Amdahl's Law, we can make informed decisions about how to best utilize parallel processing to improve program efficiency and execution time.

References

- https://www.geeksforgeeks.org/thread-in-operating-system
- https://www.geeksforgeeks.org/multiprocessin