PROJECT REPORT

CS3103

Group 3	Name	SID	Email
Member 1	HUNG Yiu Hong	56662696	yiuhhung3-c@my.cityu.edu.hk
Member 2	LEUNG Tak Man	56275395	takmleung2-c@my.cityu.edu.hk
Member 3	CHAU Yee Lee	56441513	leeychau3-c@my.cityu.edu.hk

Contents

Introduction

This report presents the solution designed for the parallel character counting. Linux pthread, mmap are implemented to speed up the counting. We also use the pthread_mutex_lock to solve the thread synchronization problems.

With the use of threads, parallelism can be achieved. By running multiple threads in parallel over different CPU cores, we may speed up the running process. Hence, tests were made to find the optimal number of threads for our program in order to have the most ideal execution result. Contents of the file have been divided into parts and handled by multiple threads.

In order to prevent data inconsistency, pthread_mutex_lock would be used in our project. Pthread_mutex_lock could lock the critical section, then no other processes can be executed in the critical sections. Only one process can access the critical section at the same time. It is a powerful tool to solve synchronization problems.

Implementation

Overview of the Program's Logic

The proposed program generates multiple threads to count characters in parallel. The structure of it can be divided into several parts (in execution order):

1. File processing

The program will read inputs from arguments, scan for file paths if there is a directory, perform sorting, and store file contents in an array.

2. Thread configuration

The program will decide an appropriate number of threads to use, then launch and wait for the threads.

3. Thread execution

When the program creates a thread, the thread will loop through all files, perform counting, then add up the results.

4. Result display

The program will display the results after the completion of thread execution.

Accessing Input Files

We use mmap() to access input files since mmap() is more efficient than traditional file accessing approach such as fread(), fgetc(), fscanf()... Compared to them, mmap() does not consume large memory like fread() even if the file is large. In contrast, it consumes address space, mmap() can process up to 2 GB of a single file even if the memory available of a system is below 2 GB. Moreover, the I/O time of mmap() is much shorter, it significantly boosts up the overall performance.

For the complete flow of processing input files, consider the following pseudocode:

```
//This is not the full flow, please refer to the source code for details
begin
Global var. fileLS[65536] //must define maximum files as array is not a dynamic data structure
…
main(let c be the number of path inputs):
…
    for loop i = 0; i < c; i++:
```

```
                if path[i] is a file then:
                        add path[i] to fileLS
                else:
                        get and add all files to fileLS from path[i]
        sort fileLS in dictionary (lexicographic) order
    …
    end
```

Designing Threads

In the program, the number of threads decided is depending on the system's available CPU cores. Using all the CPU cores should let the counting process be at the most efficient level.

To support the assumption above, an experiment is done in the CSLab SSH gateway server to examine the efficiency level by comparing different numbers of threads used.

To conduct the experiment, the command "make test" is used, which tests the program with all the given test cases.

The test below shows the relationships between the max execution time and number of threads used:

Number of Threads	**Maximum Execution Time in Single Test (approximate, unit in seconds)**
4	0.832-1
8	0.486-0.657
16	0.271-0.353
32	0.197-0.262
64	0.182-0.213
96 (max available)	**0.167-0.233**

Note: Each test with different number of threads is to be repeated 3 times.

From the observation, using all the available CPU cores might result in the best performance. Therefore, in the program the function get_nprocs() is used to determine the number of threads to utilize.

Parallelizing Character Counting and Results Merging

Parallelizing the counting of characters can make the scanning task much faster than sequential counting. In this project, for each file, the contents of the file will be divided into multiple parts, multiple threads will count part of the contents concurrently.

Before creating the threats, the length of the file contents and the number of threads to generate need to be determined first in the main() function (details will be covered in later sections).

Regarding to the flow in each thread, the thread will loop through all files given. It will compute several data first, then perform counting process for each file.

To acquire the important data mentioned above, a formula is used first to compute the length of the contents each thread counts.

$$len = \frac{TLFC}{T} + 1$$

Where len = length of the contents each thread counts, $TLFC$ = total length of the file contents (excluded '\0'), and T = the number of threads to use.

If $TLFC$ is strictly smaller than T, then only the first thread will count the characters (i.e, len is equal to $TLFC$).

Then, another formula is used to compute the starting indices for each thread (where to count characters in the whole file content).

$$SI = TID * len$$

Where SI = starting index and TID = thread ID (start from 0), TID is unique.

The calculations above are to be done in the threads separately, the counting process will start afterwards.

Concerning the counting process, the thread will loop through part of the content in a file, then store the counting results for each character in a temporary array.

Meanwhile, in each thread, when a thread completed the counting process, it will merge and store the results, a global variable int sumResult[n][26] is prepared for the threads to store their counting results (n equals to number of files). Each thread must

acquire a lock before modifying the data in that global variable to avoid data inconsistency.

Functions Used

The functions used in the program are as follows:

void * worker(void * cfg)

This is the threat function used to count characters in parallel. The argument is a struct that contains only the thread ID. This function is called by the main() function.

void sortFile(char fileLS[][512], int flen)

This function is used to sort the given file list in dictionary (lexicographic) order. It is called by the main() function.

int isFile(const char *path)

This function is used to check whether the given path is referring to a file. If so, it returns true, vice versa. This function is called by the main() function during the file path scanning process.

void processDir(const char *name)

This function is used to recursively scan and add the files to the file paths list. It is called by the main() function at the first time during the file path scanning process and called by itself if more directories are discovered.

Experimental Results (Evaluation)

Correctness (Tested on 25 November 2021, 09:53, server's CPU load was ~20%)

Attempt 1	Attempt 2	Attempt 3
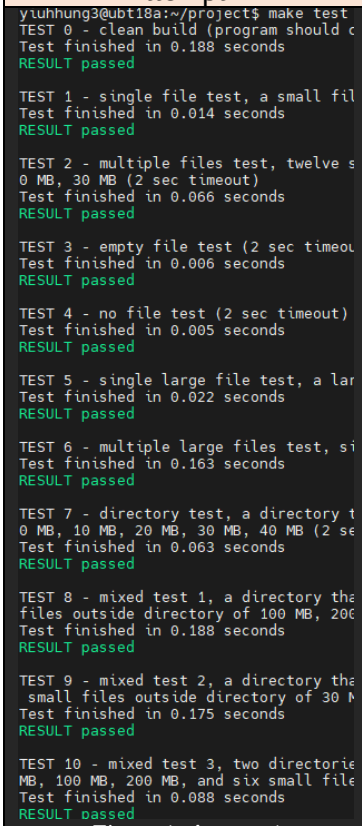	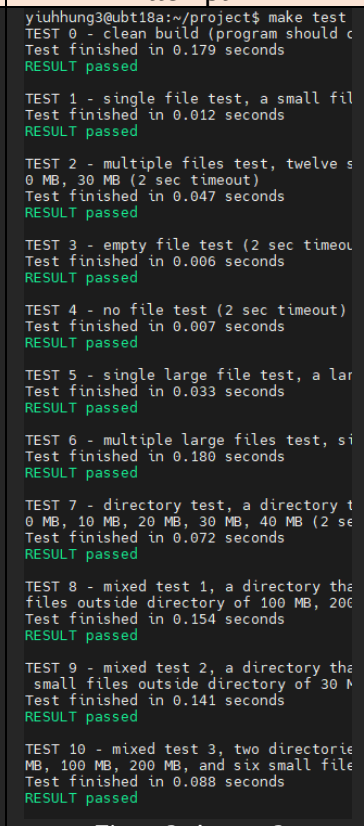	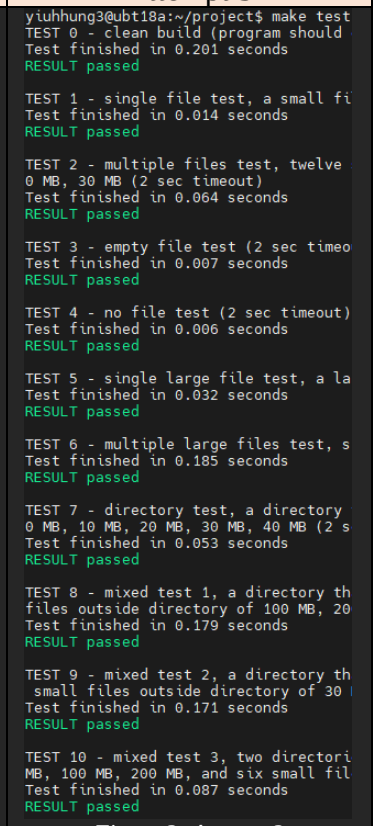
Figure 1: Attempt 1	*Figure 2: Attempt 2*	*Figure 3: Attempt 3*

Performance

The average speed of our implementation (calculated with the results above):

Test 1	Test 2	Test 3	Test 4	Test 5	Test 6	Test 7	Test 8	Test 9	Test 10	Total
0.013	0.059	0.006	0.006	0.029	0.176	0.062	0.173	0.162	0.088	0.776

Expected results under the section "III. Project Guidelines" in the project instruction document:

Test 1	Test 2	Test 3	Test 4	Test 5	Test 6	Test 7	Test 8	Test 9	Test 10	Total
0.038	0.085	0.014	0.006	0.053	0.278	0.084	0.359	0.285	0.157	1.359

From the results and observations, in both single and multiple file tests, it is discovered that in general, the performance of our program is better than the baseline statistics provided. This might indicate the success of developing parallel processing mechanisms with data consistency protection.

Difficulties Encountered

At middle stage of the development, when we discovered it is possible to assign part of the file contents to multiple threads for counting, in main() function, during the process of thread generation with loops, we tried to parse the substring of the output from mmap() to the threads as an argument input using strncpy() / memcpy(). However, somehow the threads could not read the data from its argument correctly, in the threads, the string received turned into binary contents. This problem consumed most of the time for debugging.

It is solved by changing the way of storing mmap content outputs (make it globally accessible) and the input arguments needed for each thread (only information about where to start counting and the length of counting).

Furthermore, in order to reduce the context switch time, some more complicated approaches are attempted. For instance, one of the methods is to let the threads loop through all files and use mmap() to read only part of the data (the previous version of current mechanism). Consider the following pseudocode:

```
begin
global var. finalResults[n][26] //where n = total number of files
global var. fileLS[n][512]

Thread:
        int subResults[n][26] //stores character counts
        for loop i = 0; i < n; i++:
                open file, get file length (file path refers to the i th index of the file list)
                string contents = mmap() the file, read and return part of the content
                perform counting process, modify subResults[][]
        end of for loop
        acquire_mutex_lock()
        add subResults' count to finalResults[][]
        release_mutex_lock()

main():
        …
        print finalResults[n][26]
end
```

Unfortunately, the method cannot solve the problem and return expected outputs but throws segmentation errors. From debugging, it is observed that the threads were not reading the same number of bytes even though the byte mapping length set for each mmap() in the threads were the same.

It was solved by creating a globally accessible char array and preloading the content of all the files in that array before the threads start and adjusting the corresponding computation approaches slightly.

At the late stage, when everything is working, we came across with a new challenge. The performance is fine in handling multiple files, but it was a disaster in handling single file. For instance, a single file (10MB) could consume up to 0.1 seconds to run. This problem also consumed quite a lot of time.

Against the problem, we suspected that part of the code consumes a lot of static computation time to finish. Therefore, we came up with a solution, that is commenting different parts of the code to inspect the execution time. After several attempts, the root problem had been figured out.

Originally, in each thread, the field subResult[n][26] (refer to the previous pseudocode) had a fixed size of 65536. It is discovered that a large array could increase a lot of time in accessing the data (dramatically longer traversal time). Therefore, the new size of the array is dynamically set based on the number of files detected.

Gains from the Project

The project is beneficial to drilling the thinking of the development of threads synchronization and efficiency optimization. We have learnt various concepts in the field of efficiency optimization using threads and advanced file accessing method.

First, we learnt the importance and considerations in designing threads. After the implementation and experiments, the relationships between threads and efficiency are discovered.

Second, the prevention approaches and side effects of data inconsistency protection are also learned.

Third, we learnt other kinds of file content processing mechanism. In this project, we used mmap(), we understand the difference between mmap() and other common file processing approaches.

Furthermore, troubleshooting skills are also enhanced during the project. This project emphasis on efficiency, to bring efficiency to the next level, it is common to derive

many subproblems for us to deal with. Therefore, troubleshooting strategies should be implemented frequently in the project, thereby surpassing the limits of efficiency.

Last, core concepts in operating systems aside, the project strengthened our knowledge in C programming and its applications. Before the start of the project, most of us were inexperienced in writing C language. Fortunately, this project let us familiar with one of the most popular and practical language C, thereby enhancing our competitiveness.

Contributions of Each Member

The task allocation table is as follow:

Member / Task	HUNG Yiu Hong	LEUNG Tak Man	CHAU Yee Lee
Programming	✓	✓	✓
Research: Multithreading	✓	✓	✓
Research: Accessing input files		✓	✓
Research: File paths and input processing	✓		
Research: Threads and efficiency	✓	✓	✓
Program's continuous development & inspection	✓		
In depth result testing	✓	✓	✓
Writing report	✓	✓	✓