Project 1 Report

Name: Ciel Student id: 7

Contents

1 Introduction

1.1 Project Background and Objectives

The purpose of this course design project is to enable students to initially master the methods of C language operating system design and consolidate the knowledge learned in the operating system theory class through practice. The specific goals are as follows:

i) Use C language to design and implement some basic system programs, and have a preliminary understanding of operating system development methods.

ii) In-depth understanding of processes and threads, and proficient use of Linux operating system system calls to achieve process control and management.

iii) Study concurrency and parallel mechanisms, to understand their implementation methods, performance and other characteristics.

iv) Use visualization tools for data analysis

1.2 Key Technologies Involved

Our key technical details include: Linux process management, Socket communication, POSIX threads and synchronization.

1.3 Report Structure Overview

First, we briefly explain the content and requirements of the three sub-problems of this project. **Then**, we introduce the design ideas and specific implementation and optimization methods based on these three problems. **Finally**, we have done extensive test on the performance of our program. We will show the test results and analyze the them in detail.

2 Task Description

2.1 Overview of Tasks

This project includes three sub-tasks: file copy, shell program and Multithreaded merge sort. To complete these tasks, we are required knowledge of various aspects, including Linux process management, pipe communication, data transmission protocol, Linux system call, Linux multithreaded programming *etc.*

2.2 Detailed Requirements

2.2.1 File Copy

In this task, we are required to deign C programs to copy the text content from a file to another. The concept of ”copy” is reading data from a source file and writing it to the destination file. The basic design (MyCopy.c) is to proceed the entire work in one process. In an advanced design (ForkCopy.c), the parent process should spawn a new process and execute the copy task in the new process. The ultimate design (PipeCopy.c) requires the program to spawn 2 child processes. One child process reads data from the source while the other one write data to the destination. The 2 processes communicate through a pipe.

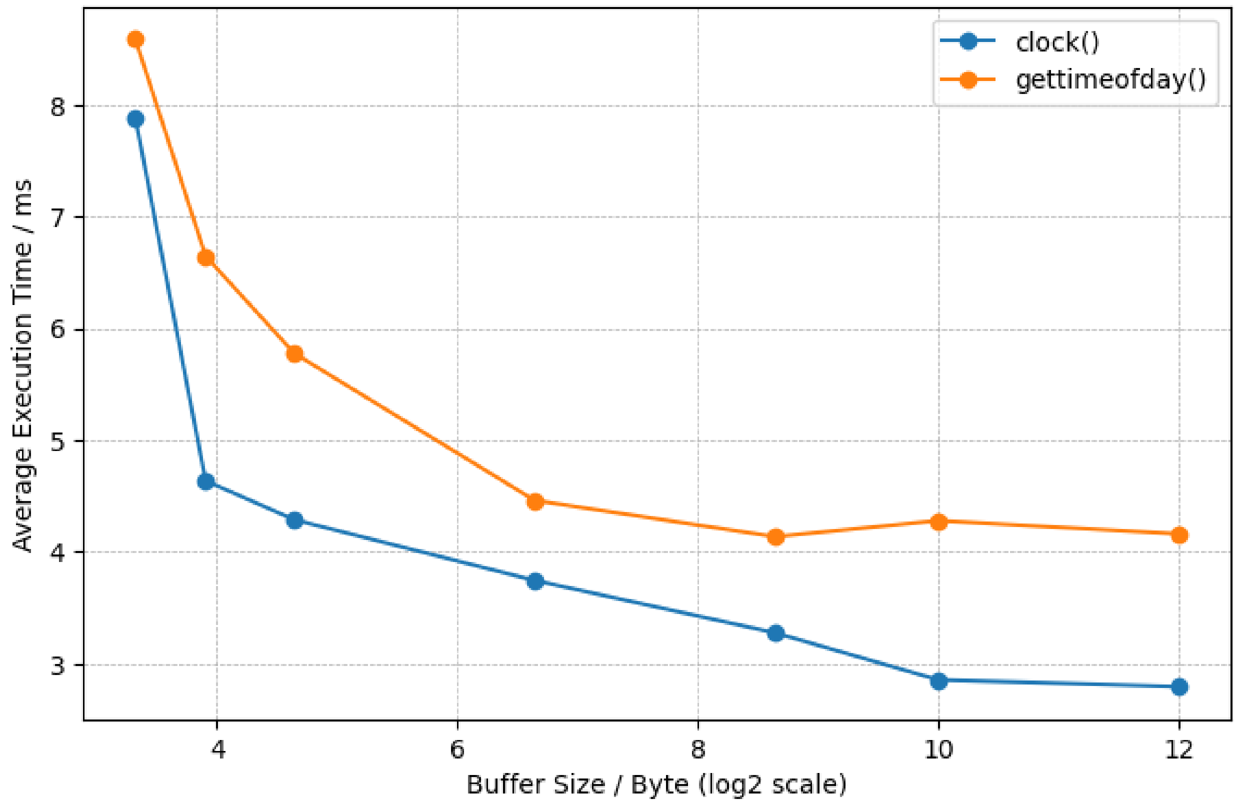


Figure 1: The experimental results of the Mycopy method under two time measurement approaches: clock and gettimeofday. The file size used in the experiment is 1.1MB.

2.2.2 Shell Program

First, we are required to write a shell-like program that can handle commands with multiple arguments. Specifically, it should correctly handle pipe command and 'cd' command. Then, we are required to implement the shell program as a server, which can support multiple clients and communicate with them using Socket.

2.2.3 Multithreaded Merge Sort

In order to implement multi-thread merge sort, we should first design a vanilla merge sort porgram (MergesortSingle.c) and upgrade the program to support multithreading (MergesortMulti.c).

3 Design and Implementation

3.1 File Copy

In the basic design, we want to read data from the source file and write it into the destination file. At most of the time, we are not able to read the entire file at once, as the data might be far more larger than the main memory. So we apply a data buffer, which can be accommodated by main memory, to transfer data from source to destination. Through separating the whole process, we are able to copy large file. Given the substantial size of the file to be copied, the probability of glitches arising during the copying process cannot be overlooked. It is necessary to set checkpoint to detect

Buffer Size	10B	100B	1KB	4KB
MyCopy	129.77034	72.146519	61.538940	66.757025
ForkCopy	49.231791	30.564869	25.847379	25.262132
PipeCopy	178.78939	55.304633	32.750605	29.457934

Table 1: The growth rate of copying time for file sizes ranging from 1KB to 16MB under various copying methods and buffer sizes.

whether the process runs correctly. There, I get the return value of I/O functions to check if the amount of data read or wrote equals my expectation. But one problem I meet is that there will be unexpected extra content at the end of the destination file after copying. It is because when the last block of data in the source file cannot fully fill the buffer, the previous data in the buffer will not be overwrite and remain in it, causing unexpected situation. To solve this problem while considering the design simplicity, we clear the buffer each before reading.

After implementing the basic design, the implementation of fork copy becomes relatively simple. We just need to invoke the system call of fork to spawn a new process and use exec system call to let the child process execute the above program. What the main process should do is to wait for its child process to complete. It is noteworthy that, in order to enhance the robustness of the program, it is imperative to verify the success of each system call after its invocation. In the event of a failure, it is crucial to accurately log error messages, thereby assisting programmers in pinpointing the source of the malfunction.

The creation and use of pipe is the essence of the pipe copy program. We use pipe system call to create a pipe and use open, close system calls to manipulate specific ports of a pipe. It should be clarified that open and close system calls are not exclusive to pipe, but can manipulate all file descriptor. Other files can also be opened or closed by these system calls. Within an ordinary pipe, data is constrained to flow in a singular direction, ensuring an unidirectional and orderly transmission of information. In our program, the child process of reading data should close the read end and open the write end while the other one closing the write end and remaining the read end. As parent process has no need to communicate through pipe, it should close its both ends.

3.2 Shell

To achieve the fundamental functionalities of a shell, our program must be capable of parsing user command lines, handling system calls, and executing the corresponding commands with precision. Basic command line parsing can be accomplished by splitting the input string using spaces and subsequently reconstructing it into an array of parameters. Following this, the essential command line functionality can be achieved by spawning a child process and utilizing the execvp function within that child process to execute the desired command.

Pipe command indicates that the output of the previous system call will be redirected to the input of the next system call. To support it, we should modify our parsing method, apply pipes for inter-process communication and use dup2 system call to redirect the I/O of processes. Compared to pipe command, cd command is much more easier to support. It suffices to handle the cd command exclusively by utilizing the chdir function.

To enable our shell to operate as a server, we employ sockets. We devise a comprehensive mechanism for creating, binding, listening, and managing client connections to ensure stability and security when accessed by multiple users concurrently. There is no fundamental difference between the server version and the basic version. The only alteration is that the standard input and output have been reconfigured to facilitate interaction with the client.

3.3 MergeSort

The fundamental merge sort algorithm has been taught in algorithm course. A recursive function applied to divide and conquer the whole array. Typically, we divide the array into 2 subarrays in every recursion layer.

The multithreaded version of merge sort can carry out this ”divide and conquer” process in parallel. To achieve meltithreaded programming, we apply 2 methods. For the first method, we assume the number of available threads is power of 2. So we can simply stop threading at a specific depth. In the second one, we use a semaphore to indicate the current number of available threads. Our program create new threads when there are available threads and simply execute sequently when there is no thread available.

4 Experiments and Testing

4.1 Testing Environment

Our development and testing environment is established on Oracle VirtualBox virtual machine, running a 64-bit Ubuntu Linux operating system. The system is equipped with a quad-core processor, 4GB of RAM, and 20GB of hard disk space, which provides a robust platform for executing and evaluating our program.

4.2 Copy File Performance Analysis

4.2.1 Time Measurement Method

We compared two techniques for timing, the result in Fig. 1. It is obvious that the average time recorded by clock function is less than that recorded by gettimeofday function. This is easily explicable. The clock function measures the CPU time consumed by the program, whereas gettimeofday records the elapsed wall-clock time from the program’s commencement to its conclusion. Since a process is often interrupted by other processes, the actual execution time of the program is typically shorter than the elapsed wall-clock time.

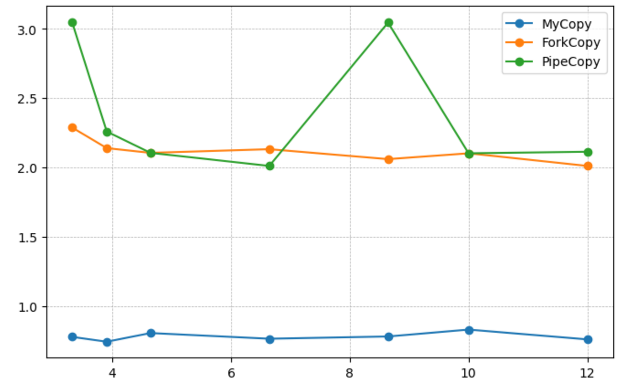
In this experiment, given that the results obtained from gettimeofday are more stable than those from clock, we have elected to use gettimeofday for timing in subsequent experiments.

4.2.2 Main Experiment

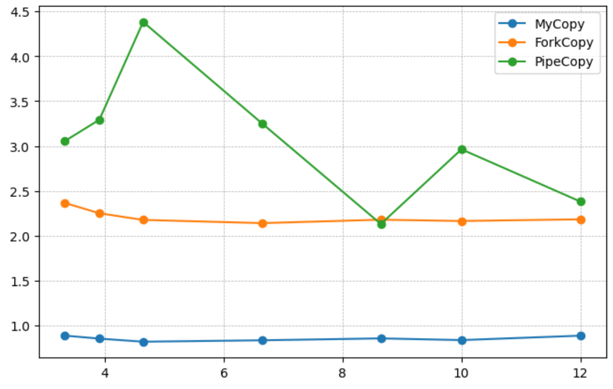
The efficiency of copying is influenced by the copying method, file size, and buffer size. We conducted extensive experiments to investigate their impact and interrelationships. We show our quantitative result in Tab. 1, and visual result in 2. Based on the experimental results, we discovered that:

i) The original MyCopy method exhibits the shortest execution time, followed by the ForkCopy method, while the PipeCopy method requires the longest execution time. We attribute this to the fact that the ForkCopy method incurs additional overhead from creating new processes but does not achieve true parallelism. In contrast, the PipeCopy method not only involves process creation overhead but also introduces the overhead of pipe communication, resulting in a longer execution time.

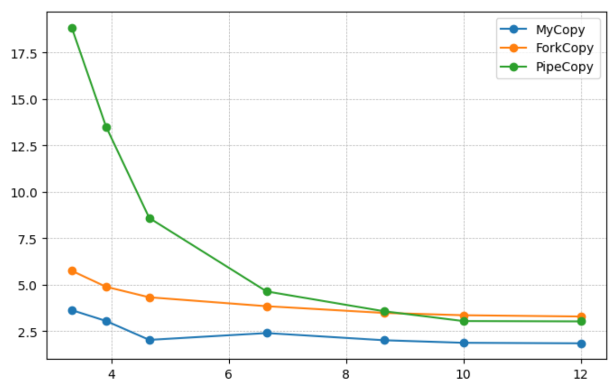
ii) As our intuition suggests, when the file size increases, the execution time for copying also rises accordingly. However, the rate of increase in execution time does is smaller than the rate of growth in file size. This indicates that, apart from file size, other factors such as system call



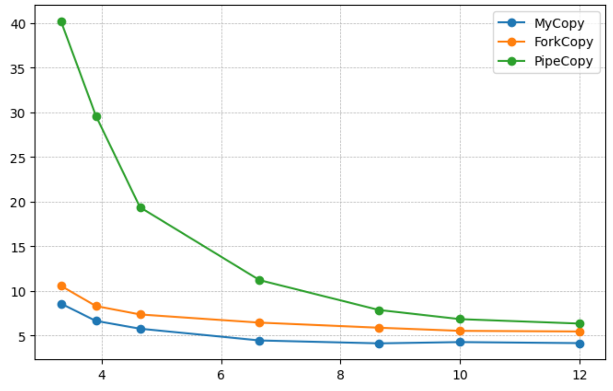
(a) 1KB



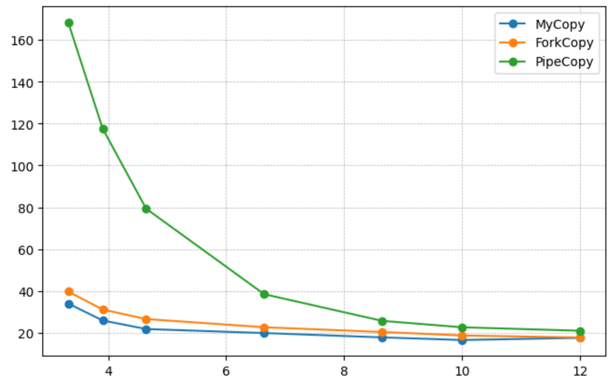
(b) 22KB



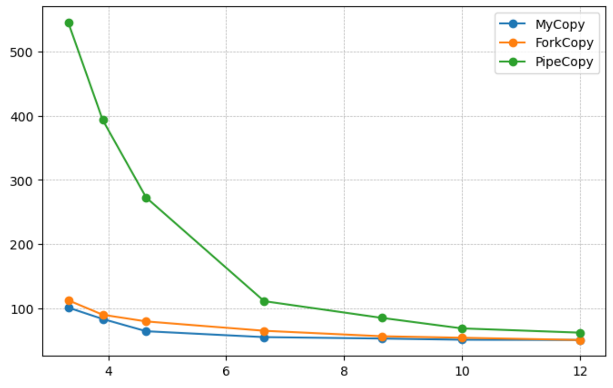
(c) 482KB



(d) 1.1MB



(e) 4.7MB



(f) 16MB

Figure 2: Average running time under different experiment settings. In every figure, the Blue Line, Orange Line and Green Line indicate the running time of MyCopy, ForkCopy and PipeCopy, respectively. X-axis represents $\log_{10}$ of buffer size (Byte). Y-axis represents running time (ms).

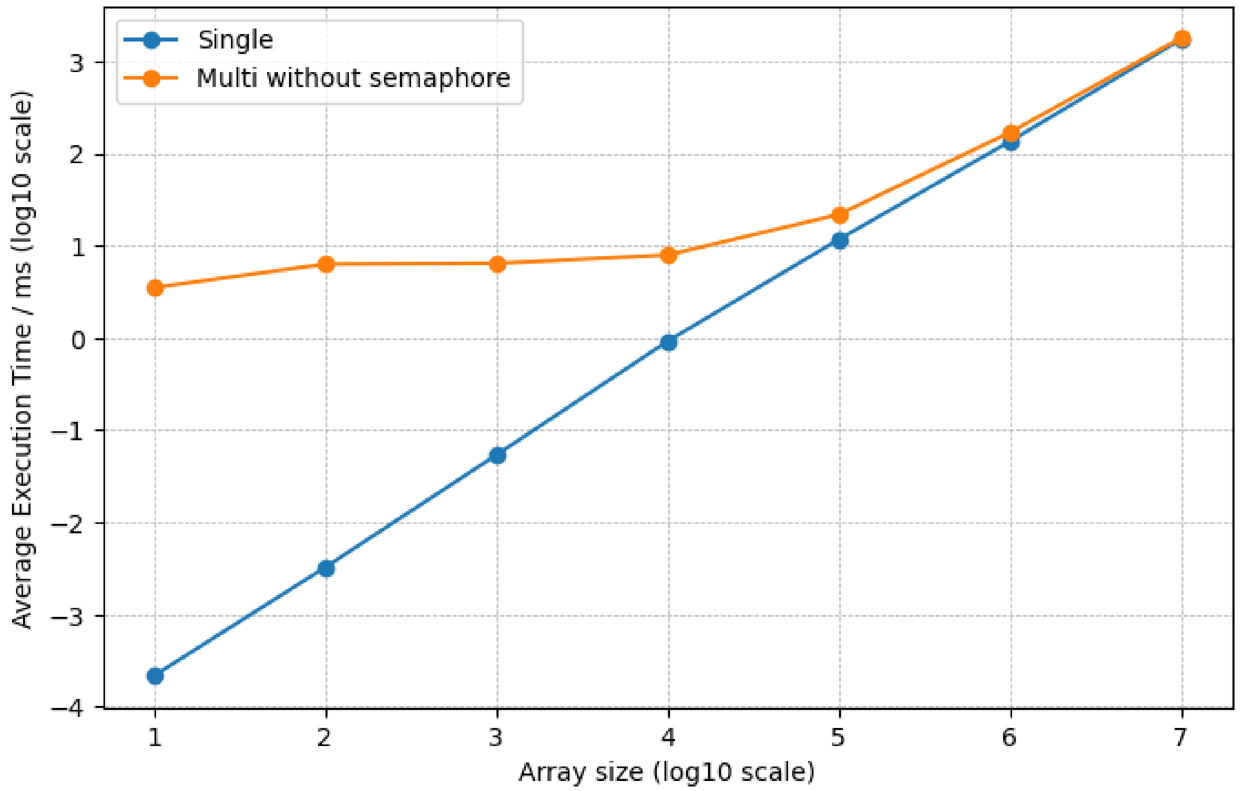


Figure 3: Comparison between single-thread programming and multithreaded programming. The number of available threads is 16.

overhead and hardware performance also influence copying efficiency. We found that the time consumption growth rate of the MyCopy method is greater than that of the other two methods. This aligns with our hypothesis, as the other two methods involve additional overheads from process creation and pipe communication. Consequently, the proportion of overhead solely attributed to I/O operations is smaller in these methods.

iii) When the file size is big ($\geq$ 482KB), the larger buffer size results in shorter running time. But when file size is relatively small (1KB, 22KB), it is difficult to discern any significant impact of buffer size on copying efficiency. The impact of buffer size varies across different copying methods. In the PipeCopy method, the influence of buffer size is significantly more pronounced compared to that in the MyCopy and ForkCopy methods. Furthermore, for all methods, as the buffer size increases, the marginal performance improvement diminishes.

4.3 Merge Sort Performance Analysis

4.3.1 The Impact of Multithreaded Programming

To investigate the impact of multithreading on the performance of merge sort, we conducted a comparative analysis of the execution times of single-threaded merge sort and multithreaded merge sort without semaphores across datasets of varying sizes. We show the result in Fig. 3.

in our experiments, the performance of multithreaded programming did not surpass that of single-threaded programming. This may because our multithreaded implementation is too naive.

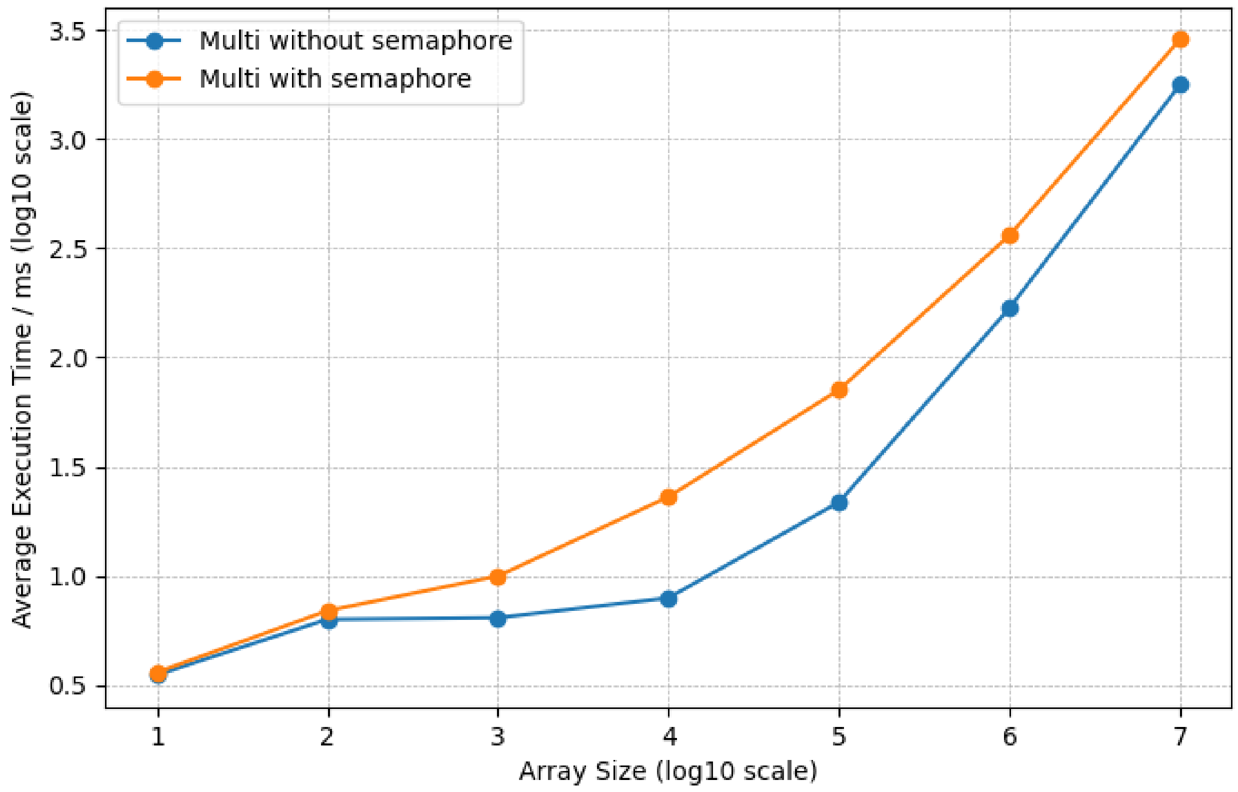


Figure 4: Comparison between depth based thread allocation and semaphore based thread allocation. The number of available threads is 16.

The performance gained from multithreading is outweighed by the overhead incurred in creating threads.

Nevertheless, we can still observe a significant impact of multithreaded programming. In single-threaded merge sort, the running time increases linearly with the size of the dataset, whereas in multithreaded merge sort, the running time exhibits non-linear variations when the array length is relatively small ($\leq 10^4$Bytes).

However, when the array is very large ($\geq 10^5$bytes), the running time of the multithreaded approach once again increases linearly with the size of the dataset. This is because our thread count is limited, and the vast volume of data overshadows the effects of multithreading.

4.3.2 The Impact of Semaphore

We designed two multithreaded merge algorithms. One creates threads based on depth, while the other creates threads using semaphores. We compare their performance and show the result in Fig. 4.

Initially, we hypothesized that the method using semaphores to create threads would perform better, as this design is dynamic and could better achieve parallelism. Contrary to our initial assumption, the method of creating threads based on depth proved to be superior.

Upon analysis, we formulated a hypothesis. We can envision the merge sort process as a binary tree, where creating threads based on depth would preferentially generate threads for nodes closer to the root. These nodes undertake a greater share of the computational workload in merge sort,

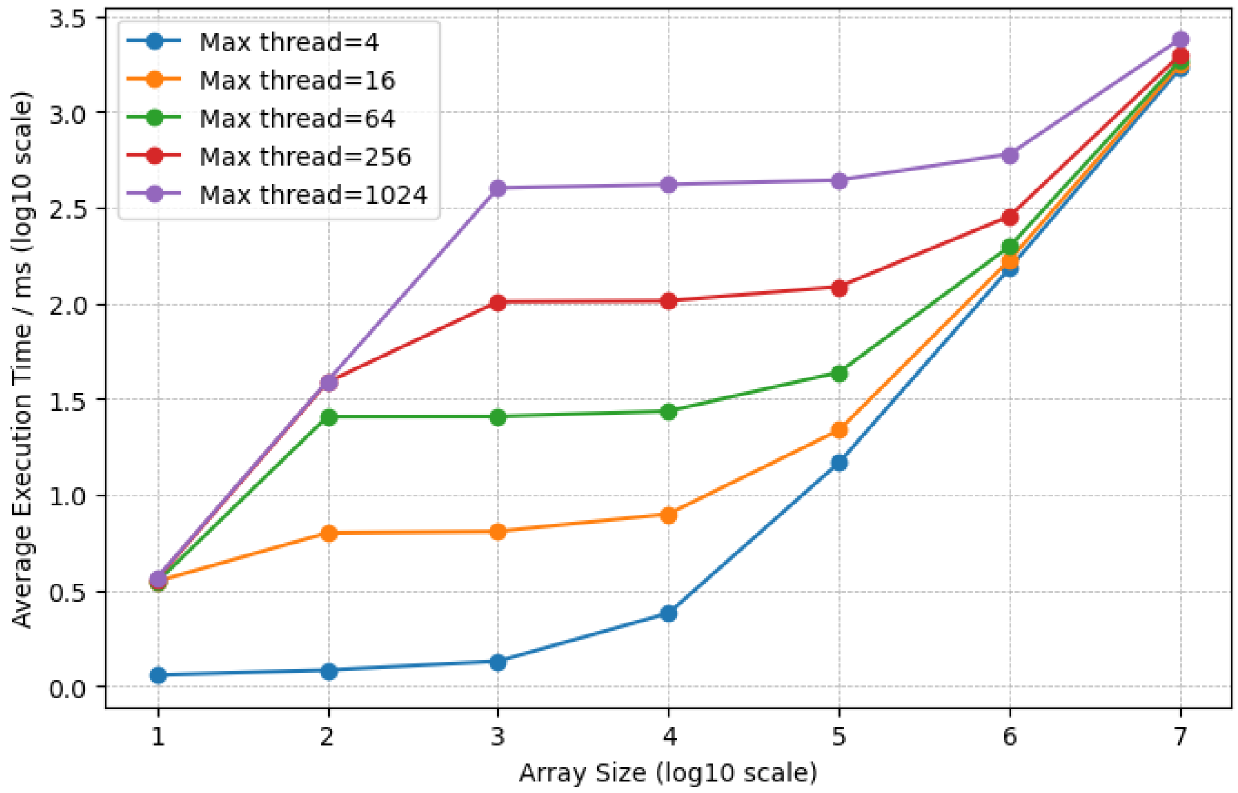


Figure 5: Visualization results of the running times of depth-based multithreading across different data volumes and available thread counts.

thereby fully leveraging the benefits of parallelism and enhancing overall performance. In contrast, the method of creating threads using semaphores might generate threads for nodes closer to the leaves, thereby preempting thread resources from more critical nodes and consequently diminishing overall performance.

4.3.3 Main Experiment

In the main experiment, we selected the merge sort method that creates threads based on depth and measured the program's running time across different thread counts and dataset sizes. The result is reported in Fig. 5. We observed that for all thread counts, there exists a range of dataset sizes within which the program's running time remains nearly constant. Moreover, the upper bound of this range increases with the number of threads. This observation aligns with our hypothesis in 4.3.1. As the number of threads increases, the influence of multithreading becomes harder to obscure by the volume of data.

5 Conclusion

In this section, we summarize the completion of the project, highlighting the key aspects of the implementation and the overall outcomes. The project provided hands-on experience with Linux system programming, including process management, inter-process communication, and multithreading techniques. Through the development of file copy utilities, a simple shell, and a multi-threaded

MergeSort implementation, we gained a deeper understanding of concurrency and system calls.

This project reinforced our ability to write robust and efficient C programs for concurrent execution, which is a crucial skill in system-level programming. Future improvements could involve more advanced performance optimizations, enhanced error handling, and exploring alternative concurrency models such as asynchronous I/O or parallel computing frameworks.