Project 1 Report
Name: xiaoyu_ma Student id: 523031910469

1 Introduction

In this Report, we implement some system's programs using C language, which consist several system call and put some concepts of system design into use. Such as pipe, multi-threading and multi-processing, socket and so on.

Here we analyse the performance and break down the implementation details. Including some Interesting findings or issue

2 Design & Implementation

In this section, we breakdown each sub projects into its design and a overall analyse on its implementation

2.1 Copy

This sub project requests us to Write C programs to copy the text content of one source file into a destination file, which is achieved in several small steps. As well as implement the copy with Fork or Pipe.

2.1.1 MyCopy

This is the most plain part of the requirement, Just simply do the copy. But to ease the pressure when performing benchmark, I extend the arguments numbers to 3 to enable convenient assignment of buffer size when copying.

the execution command works as follows:

```
./MyCopy (or ForkCopy, PipeCopy) <InputFile> <OutputFile> <Buffer Size>
```

In implementation, I found that the buffer Size plays a very important role in the velocity of copying, but it must appears as the exponent of 2, otherwise the copying speed won't be affected

The evaluation details will be discussed later

Meanwhile, the copy buffer must be initialised in the way of malloc(), it can't be directly declared by defining a char Array with bufferSize (i.e. char buffer[BUFFER_SIZE]). The possible reason might be that the space for a array is pre-allocated in program, change the length cannot change the overhead of transferring arrays

2.1.2 ForkCopy

This part simply calls execl() linux system call to execute the MyCopy in another Process. The key point is to call execl() in another process. Thus introduce extra overheads and result some difficulties when timing the program

2.1.3 PipeCopy

This part we Write a C program that forks two separate processes: One for reading from the source file and the other for writing into the destination file. These two processes communicate using "pipe" system call.

In this request, we should create a pipe and connect the 2 ends with different function. The point that needs to pay special attention is to close the unused end of pipe and close the pipe after reading or writing is end to signal the finish of SubProcess.

2.2 Shell

In this section, we implement a shell in linux, which execute user inputed system commands, and Write the above shell program as a server. When the server is running, clients are allowed to connect to the server and run the commands. Noted that I am not allowed to use 'sh -h'. The tasks can be break down to 2 parts: Shell Core and Server Core

2.3 Shell Core

The projects requests us to support cd and pipe notation in CLI, hence we break the project into the following parts:

2.3.1 Parse the input

Firstly, we need to parse the input to the structure that is easy for us to manipulate. Because the input of a command is finished in a single line, we first parsed the string by | , and parse each substring by space. After that , we get the detailed information of each command.

2.4 Implement ' | '

Next, we execute each command separated by ' |' in sequence. Since the ' |' takes the output of previous command to the input of following command, we use 2 pipes:

- One to redirect the output of command to a pre-defined C string using dup2()
- One to put the buffer String to 'STDIN_FILENO', which is used as the input of a command.

2.4.1 Implement 'cd'

To realise 'cd', we use chdir() call. After this is called, the followed execution is skipped , and proceed on processing next segment of command

Then the thorough program comes. Additionally, the first and last command need some special checks on input and output. We just take the output of last command as the Final results that present to user

2.5 Server Core

I firstly build a server framework with socket, then embeds the shell Core to it.

2.5.1 Parallel Client

Each time we received a connection request, the initialise the connection socket and throw the whole serve to a thread. then detach the thread and move on listening to the port

2.5.2 Support parallel Client

Because the threads work in one process, each chdir() call in process will change the external process's working directory. So we need to cache the working directory of each thread, and update them after execution, change working directory before execution.

Here, atomic operation is required, we don't want the working directory be changed by another thread in execution. So I use lock to to lock each section of command execution

In the directory, there are 2 files: 'shell.c' and 'shellSingle.c' , the former is the server version, and the latter is the single version. The single version is used to test the correctness of the shell, and the server version is used to test the server's correctness. the details running guideline can be found in the README.md

2.6 Sort

In this section, we Implement MergeSort with multithreading programming. The requirement is in 2 parts

- Implement a plain merge sort
- Modify the merge sort into form of multi thread

2.6.1 MergesortSingle

A ordinary merge sort program , nothing fancy.

2.6.2 MergesortMulti

Write the merge sort with multithreading programming. First we divide the array into several slots of equal length of :

$$slot\_length = \frac{array\_length}{Max\_threads}$$

Then perform merge sort in each slot, and merge the sorted slots with reduction method.

3 Evaluation

3.1 Copy

3.1.1 Benchmark using Clock()

In Figure 2 , we can see that the copying time consumption decreases as the buffer size grows, which quite cater our prediction. But why this phenomenon doesn't appear in Figure 1?

One possible explanation is that the clock() use CPU timestamp to calculate time, which is inherently has some errors. Furthermore, in our code, multiprocessing is used. There will be some parallelism in execution, and thus, the accuracy faded

From figure 2, the copying time consumption decreases as the buffer size grows. And it can be observed that PipeCopy use more time than ForkCopy, while MyCopy's time overhead surpasses ForkCopy.

The reason to that is pipe. Using Pipe to transmit data can be very slow, as it introduce lots of read and write operation and much scheduling overhead. ForkCopy Runs the copy program in another process, thus gains some sort of parallelism, which accelerates the process.

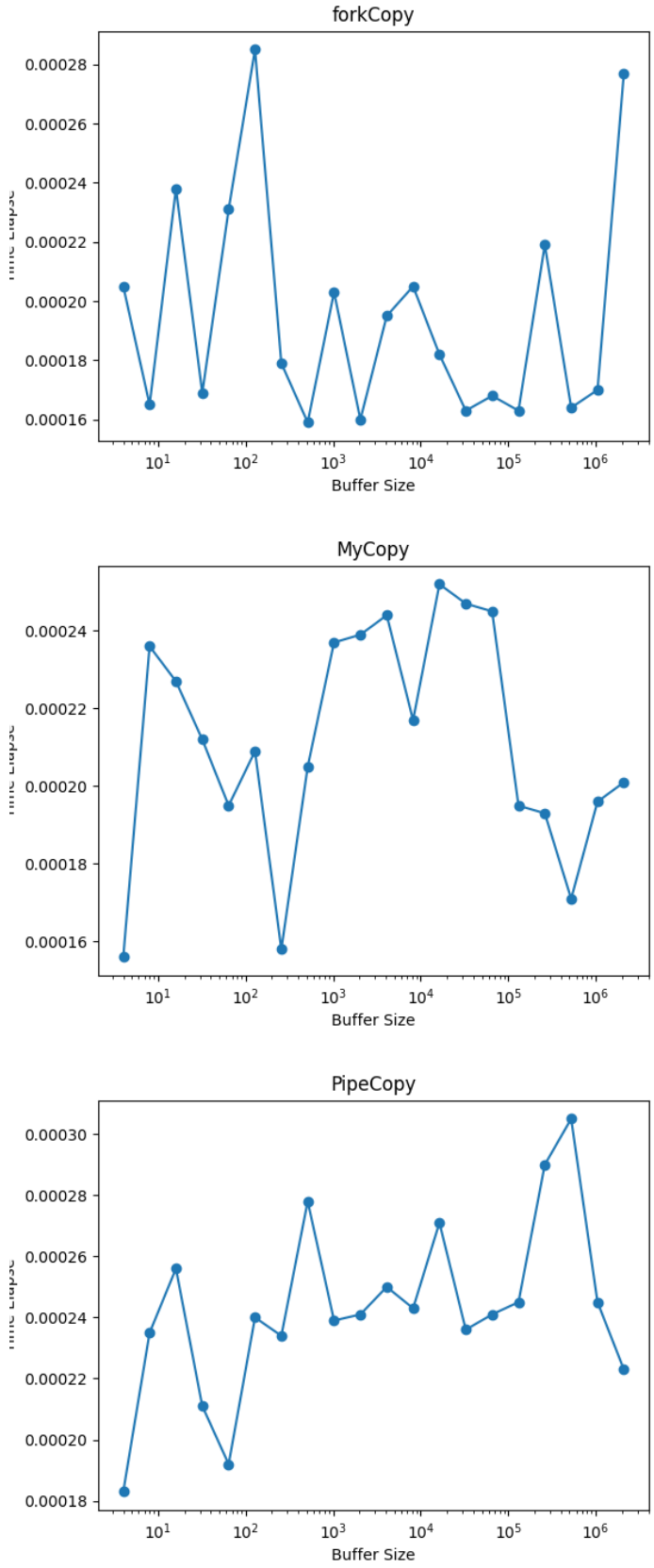

Figure 1: Using Clock to timing

3.1.2 Benchmark using Timespec and clock_gettime

In this way, we can see that with the increasing of buffer size, the time overhead of copying declines.

Compared to timing using clock(), timespec has greater accuracy. First it can calculate time in nanosecond, then it calls CLOCK_MONOTONIC to timing the program. And it won't be affected due to the multiprocessing programming.

In our evaluation, we can see some peaks in the diagram. This might be caused by the size of chunked cache mechanism inside the disk:

- Increased cost of memory allocation.
- Potential inefficiencies in utilizing the CPU cache hierarchy, leading to more cache misses.
- When the buffer size aligns with this physical block size, operations are more efficient. Conversely, if the buffer size does not match the physical block size, extra processing (like handling partial blocks or alignment issues) may occur, resulting in non-monotonic performance improvements.

In the pipecopy evaluation result, the monotonic phenomenon begins to vanish. That might be caused by the pipe in two processes. The data transmitting via pipe has more overheads, thus even if we adjust the buffer size to larger and more proper, the performance can't be enhances distinctly

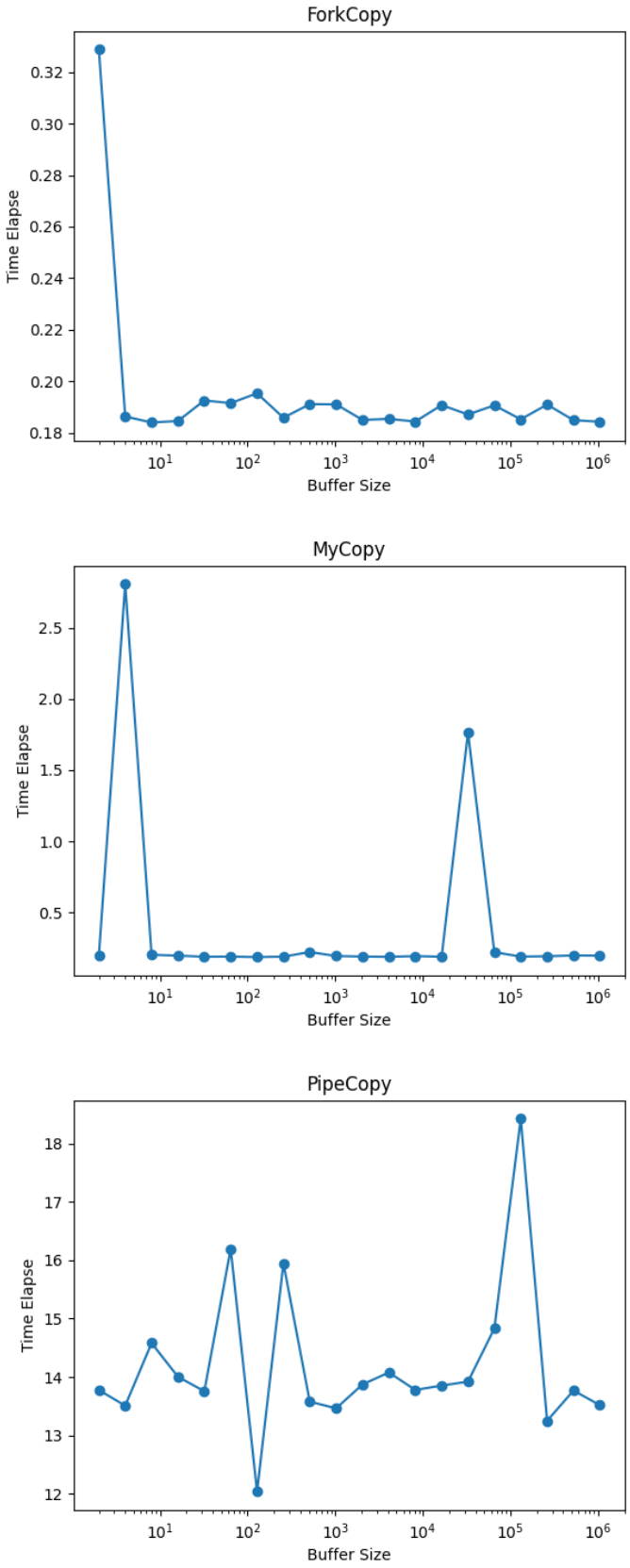

Figure 2: Using Timespec to timing

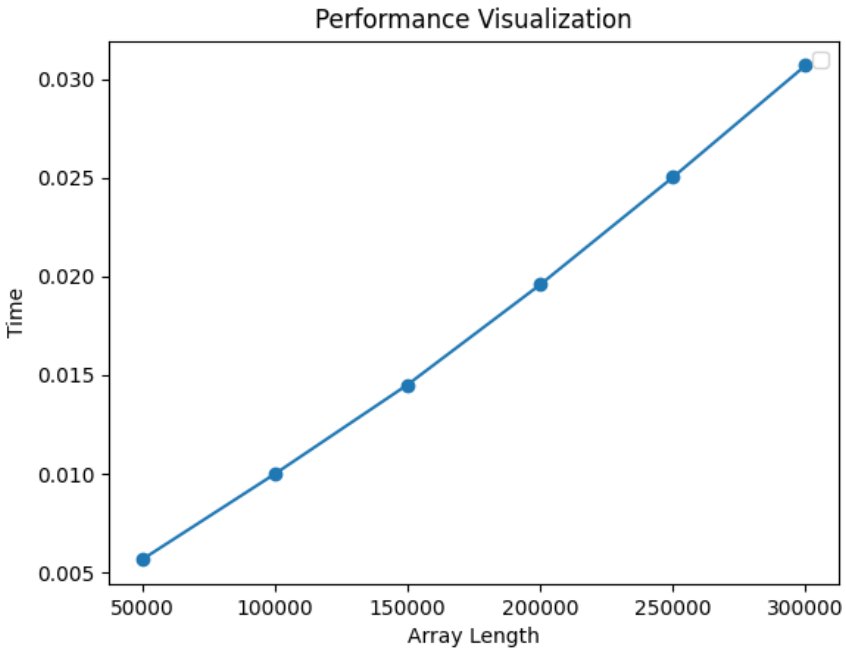

Figure 3: Benchmark MergesortSingle

3.2 Benchmark Sort

In this section, we evaluate the performance of merge sort:

- MergesortSingle : We plot the execution time with array length.
- MergesortMulti : we Compare the performance of implementations with different numbers of threads and lengths of the list. Using list length as the x-axis and use execution time as the y-axis. And distinguish different thread amount with different colours.

Note: Mergesort is a stable sort, which means its performance and time complexity will not be affected by the array given.

3.2.1 Benchmark MergesortSingle:

As we see in the graph, the time consumption increases as Array length increases. The time complexity of merge sort is $O(n \log n)$, which suggests that the line will boosts a little faster than a linear complexity. We can observed in the diagram.

1. At first, the Time increases approximately a line
2. then , the differential of the line begins to increase

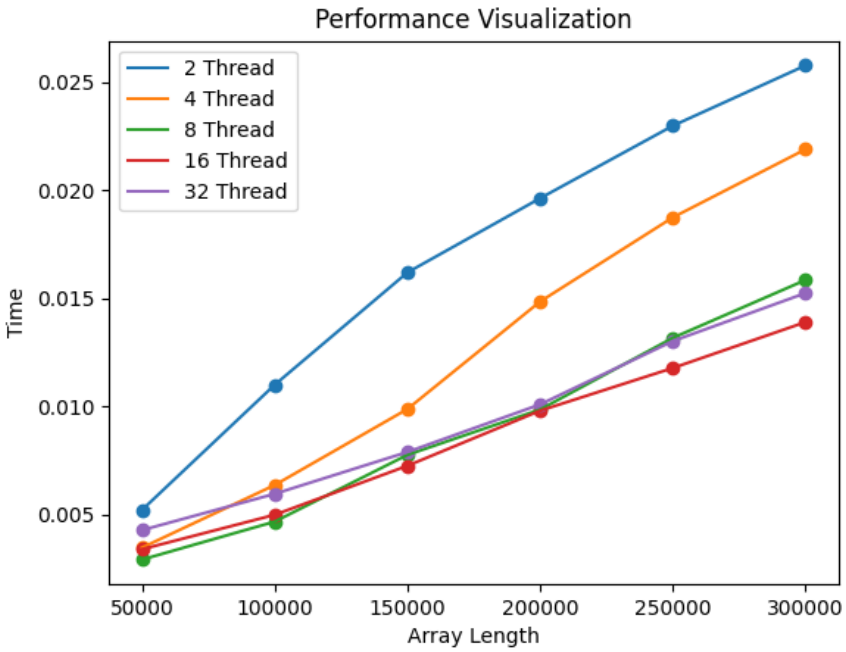

Figure 4: Benchmark MergesortMulti

3.2.2 Benchmark MergesortMulti:

As expected, the execution time increases as the array length grows across all thread configurations, And using more thread significantly improves the performance, but not always proportionally. While increasing the thread count initially helps, the performance gains become less significant beyond 16 threads.

here is the more detailed analysis

- The 2-thread implementation is significantly slower compared to higher-thread configurations.
- The 4-thread and 8-thread implementations show noticeable improvements, especially for larger array sizes.
- The 16-thread and 32-thread implementations are relatively close in performance, indicating diminishing returns beyond a certain threshold.
- When we apply 16 threads, it outperforms the condition with 32 threads. This might be attributed to extra overhead in allocating thread resources and scheduling them, And overhead in merging sorted array slots.

So, we get the optimal solution:

The best trade-off appears between 8–16 threads, balancing speedup against overhead.

Conlusion:

1. Multithreading is effective in reducing execution time for Merge Sort.
2. However, too many threads may not always lead to better performance due to overhead.
3. The optimal number of threads depends on factors like CPU core count, cache efficiency, and memory bandwidth.

3.3 Shell

In this section, some of the interesting findings will be listed:

- The method to use telnet 'ip_address' 'port' to transfer command will add extra '
r' in the end of your string, If not handled properly, it will cause COMMAND NOT FOUND issue.
Using 'nc' command does't have the problem.
- The server had to maintain the connection/disconnection information for each client, and Shut each server thread/process for their client. To simplify, I apply multithreading, but in this way, more complicated 'cd' command handling has to be done.
- A pipe can only used to connect two processes in a specific direction. That is: I need to define multiple pipes: One for receiving output of a command and the another one for sending inputs.
- When a client_serving thread is executing its command, it's better to require a lock for it. Because the threads shares a common global memory, which is used in my program to handle 'cd' command. We don't want inferences between threads.

4 Further Thoughts

4.1 Sort

In implementing Merge Sort Multi, we use many threads to sort each slot of a array. But a CPU cannot have many cores to enable full parallelism of sorting threads, additional threads offer minimal benefit.

But, **GPU** has many SMs, We may apply the sort to gpu to gain total parallelism. And get less time overhead.(while synchronization might be hard)

4.2 Shell

The shell does't support auto completion and wildcard (like *.txt). Maybe I can seek for some documents to figure out the solution

4.3 Copy

The best performance of a copy with different buffer size may differ from different computer hardware. Thus, enough knowledge had to be got for Operating System to provide efficient copy.