Project 1 Report

陈潇雨 523031910681

Experiments

The contents of 3 experiments of Project 1 and the analysis are presented below, arranged in the order of the problems.

Part 1: Copy

Introduction

The function of the copy section is to copy a file from a source to a destination. It is implemented using two processes: one handles reading from the source file, and the other handles writing to the destination file. These processes communicate via a pipe, and the user can specify the buffer size, which influences the copying speed.

- Command line

  ```
  ./Copy <InputFile> <OutputFile> <Buffersize>
  ```

Analysis

In this part of the program, I use the `fork()` system call to create two processes. The parent process is responsible for reading from `src.txt` and writing the content to a buffer via a pipe. Meanwhile, the child process reads from the pipe and writes the content to `dest.txt`. To measure the CPU usage time, I use the following formula to calculate the elapsed time in milliseconds:

```
elapsed = ((double)(end - start)) / CLOCKS_PER_SEC * 1000;
```

The performance of the program is measured by varying the buffer sizes, and the CPU time taken for each buffer size is calculated. The buffer size affects the efficiency of reading from and writing to the pipe between processes.

Test runs:

To evaluate the performance, I conducted several test runs with different buffer sizes. The results are shown in the following graphs.

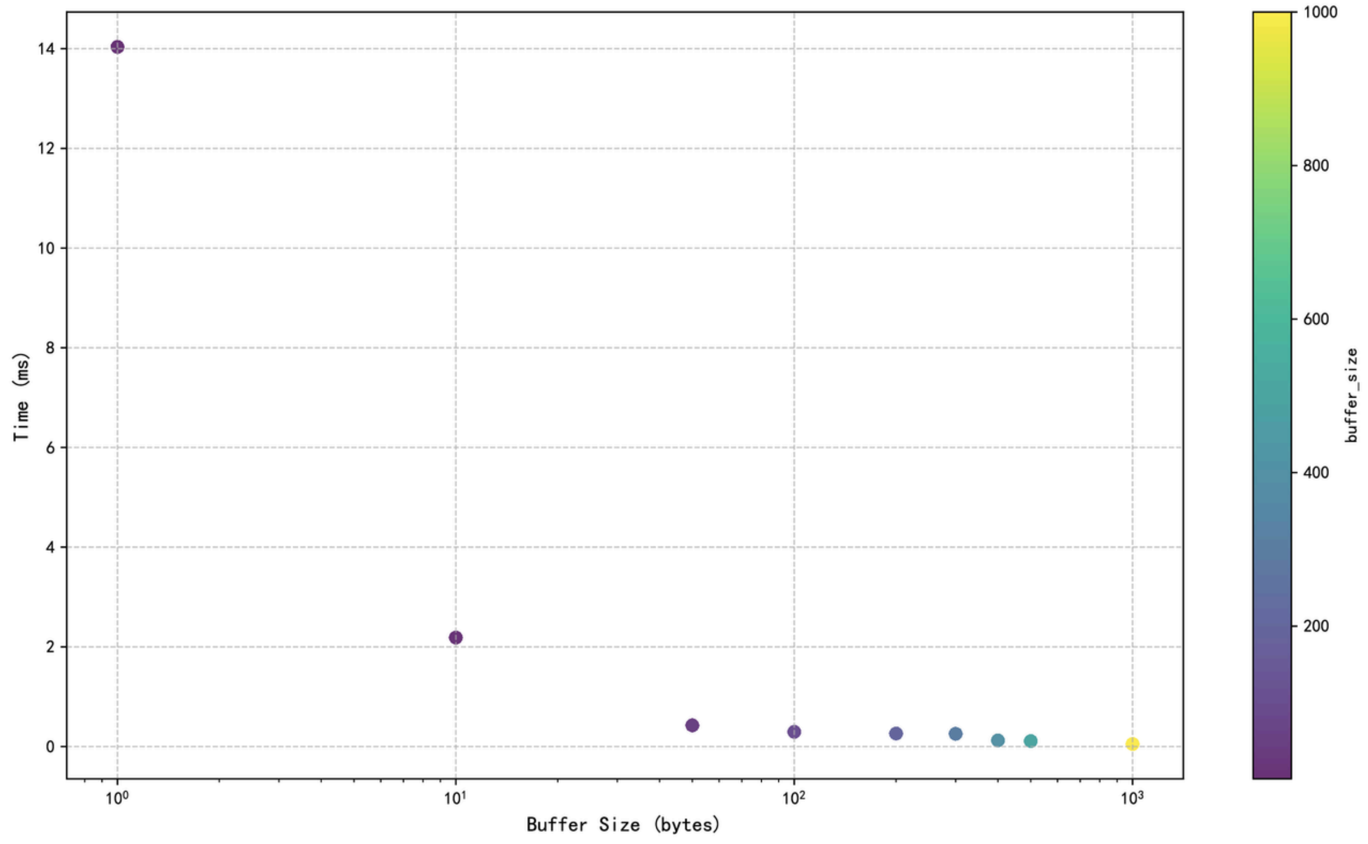


Figure 1 File Copy Time with different Buffersizes

- In my tests, src.txt (13 B) was used as the source file for the copy operation. And the pipe with the BufferSize of 1, 10, 50, 100, 200, 300, 400, 500 and 1000 bits was copied and timed respectively.

From the scatter plot, we can observe a strong negative correlation between buffer size and processing time. The relationship follows a logarithmic pattern,with processing time decreasing dramatically from 14ms at 1 bit to around 0.3ms at 100 bits. Beyond 100 bits, the performance improvements become more gradual, reaching 0.050ms at 1000 bits.

The optimal buffer size appears to be between 300-500 bits, where the performance curve starts to flatten significantly. For practical applications, choosing a buffer size between 300-500 bits would provide a good balance between performance and memory utilization.

Part 2: Shell

Introduction

The function of this project is to implement a server-side shell-like program that processes commands with arguments and pipes, demonstrating process spawning and I/O redirection using Linux system calls. The server supports multiple clients, allowing them to connect via Internet-domain sockets, execute commands, and receive results. It handles command pipelines, uses multiprocessing for concurrent client support, and includes an exit command for disconnecting clients. The server is tested using `telnet` to simulate real-world usage.

- Command line

  ```
  ./shell <Port>
  ```

- Command line for telnet

  ```
  telnet localhost <Port>
  ```

Analysis

In this part, we create a server that can connect to a maximum of five clients at a time. To achieve this, we first design a function called `parseLine()`, which splits the input by spaces and records the number of pipes. Then, we categorize the input commands into three types: commands containing pipes, cd commands, and other valid commands, while providing error prompts for invalid commands.

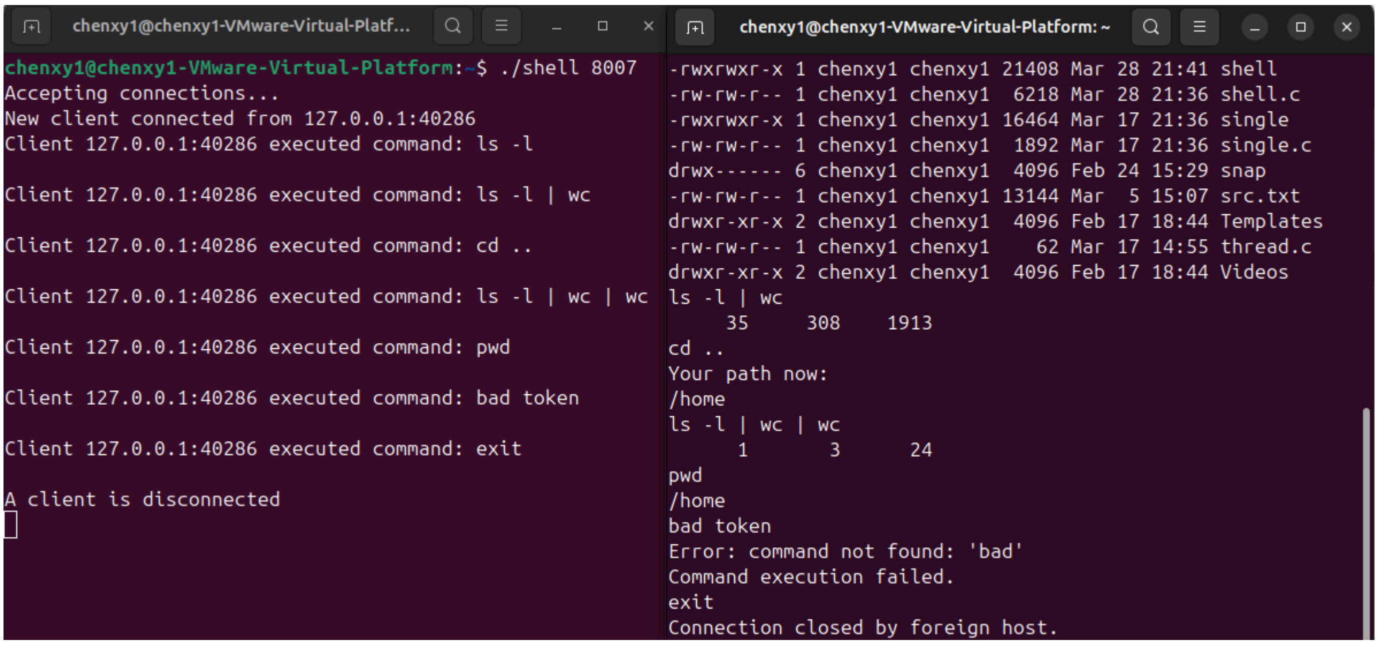
- For the cd command, we implement it through the `execute_cd()` function. When executed, it manages two scenarios: changing to the home directory (when no path is specified) or changing to a specific directory path. The function utilizes system calls like `getenv`, `chdir`, and `getcwd` to handle directory operations. For error handling, it sends appropriate error messages back to the client through the socket connection. Upon successful directory change, it retrieves and sends the new working directory path to the client, ensuring users are always aware of their current location in the file system.
- For commands containing pipes, our initial approach was to create all the required pipes in the main program at once, then spawn the corresponding number of processes, and use inter-process communication via pipes to eventually transmit the results to the client. However, since each process created a large number of pipes, some information might have been lost during transmission, and we did not see the expected results.

Therefore, we tried an alternative method by implementing this part in the `execute_pipe()` function. It recursively processes pipe commands by splitting them at pipe symbols `(|)`. For each command segment, it creates a new pipe and forks child processes: the first process writes its output to the pipe, while subsequent processes read from the previous pipe and write to the next one. The final output is redirected to the client socket. It avoids the need to create a large number of pipes at once. The results show that it works excellently.

- For commands that do not fall into the above two categories, since they can be easily implemented using `execvp`, we pass the string array processed by the `parseLine()` function as an argument to achieve the corresponding functionality.

Test runs:

The following image shows the input and output results of some instructions, as well as the error messages that occur with incorrect inputs.



```
chenxy1@chenxy1-VMware-Virtual-Platform:~$ ./shell 8007
Accepting connections...
New client connected from 127.0.0.1:40286
Client 127.0.0.1:40286 executed command: ls -l

Client 127.0.0.1:40286 executed command: ls -l | wc

Client 127.0.0.1:40286 executed command: cd ..

Client 127.0.0.1:40286 executed command: ls -l | wc | wc

Client 127.0.0.1:40286 executed command: pwd

Client 127.0.0.1:40286 executed command: bad token

Client 127.0.0.1:40286 executed command: exit

A client is disconnected
```

```
-rwxrwxr-x 1 chenxy1 chenxy1 21408 Mar 28 21:41 shell
-rw-rw-r-- 1 chenxy1 chenxy1  6218 Mar 28 21:36 shell.c
-rwxrwxr-x 1 chenxy1 chenxy1 16464 Mar 17 21:36 single
-rw-rw-r-- 1 chenxy1 chenxy1  1892 Mar 17 21:36 single.c
drwx------ 6 chenxy1 chenxy1  4096 Feb 24 15:29 snap
-rw-rw-r-- 1 chenxy1 chenxy1 13144 Mar  5 15:07 src.txt
drwxr-xr-x 2 chenxy1 chenxy1  4096 Feb 17 18:44 Templates
-rw-rw-r-- 1 chenxy1 chenxy1    62 Mar 17 14:55 thread.c
drwxr-xr-x 2 chenxy1 chenxy1  4096 Feb 17 18:44 Videos
ls -l | wc
     35     308    1913
cd ..
Your path now:
/home
ls -l | wc | wc
      1       3      24
pwd
/home
bad token
Error: command not found: 'bad'
Command execution failed.
exit
Connection closed by foreign host.
```

Part 3: Matrix multiplication

Introduction

I have written two programs: one single-threaded and one multi-threaded, to perform matrix multiplication.The single-thread program performs multiplication in the standard way, while the multi-thread program uses block matrix partitioning, with each thread handling a block. The performance of both implementations is evaluated based on the execution time, comparing the results with different thread counts and matrix sizes.

- Command line

```
,/single
./multi //When no command line argument given, the program automatically read
input from "data.in", and output to "data.out".
./multi <Size>//program will generate 2 random matrix of the given size. Both
the source matrix and the result matrix will be written into the "random.out"
```

Analysis

In this part, the file named `single.c` first reads matrix dimensions and input matrices A and B from `data.in`, then performs matrix multiplication while measuring execution time with `gettimeofday()` for precise wall clock timing. After computation, it writes the resulting matrix C to `data.out` and properly manages system resources by freeing allocated memory.

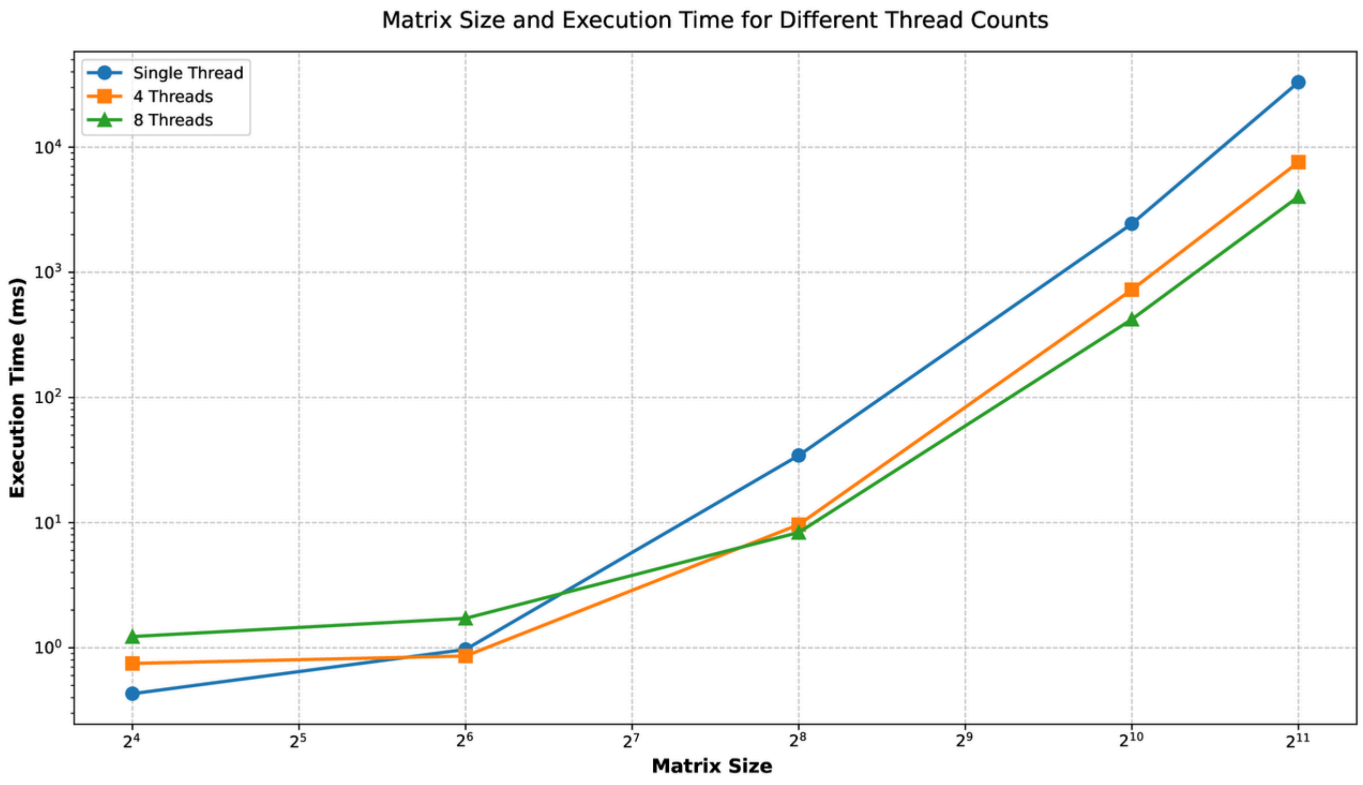
The file named `multi.c` divides the computation task among 8 threads, each handling a portion of the result matrix. The program supports two input modes: reading matrices from a file `data.in` or generating random matrices based on command-line arguments. The program measures execution time using high-precision wall clock timing through `gettimeofday()`, and saves results to either `data.out` or `random.out` depending on the input mode. It also properly manages system resources by freeing allocated memory.

- It's important to note that using the following clock-based method measures total CPU runtime, which can lead to incorrect time measurements in multithreaded scenarios. Therefore, we use gettimeofday() for timing instead, avoiding this issue.

  ```
  start = clock();
  end=clock();
  elapsed = ((double)(end - start)) / CLOCKS_PER_SEC * 1000;
  ```

Test runs:

To evaluate the performance, I conducted several test runs with different matrix sizes and thread counts. The results are shown in the following graphs.



From the graph, we can observe that as the matrix size increases, the runtime of programs with different numbers of threads shows an upward trend. When the matrix size is relatively small, programs with more threads actually take longer to run than those with fewer threads. This indicates that for smaller datasets, the overhead of thread creation outweighs the benefits of parallel computation. However, as the matrix size grows larger, the advantage of multithreading becomes evident. In such cases, increasing the number of threads significantly reduces the program's runtime.

Thus we can conclude that multithreading provides performance benefits for large matrix operations, but its overhead makes single-threaded approaches more efficient for small datasets. The trade-off depends on problem size.