OS Project1 Report

521030910229 Hao Chiyu

March 23, 2023

Contents

1 Introduction

In the first project, I finished three different labs. In the first lab, I built a pipe between father process and child process then implement Copy function by using it. Then I used visualization tool Gnuplot to display my result. In the second lab, I created a server which is able to parse the commands from client and execute them. Especially, it supports multiple-client and graceful shutdown. In the last lab, I implemented a thread pool to support multi-thread matrix multiplication. I also try some different algorithm to get the best performance. Each lab taught me something :D

2 Pipe Lab

2.1 What is ”pipe”

Pipe is a way for Linux to communicate across processes. Pipe can be divided into anonymous pipe and famous pipe and we mainly talk about anonymous pipe in this lab. Anonymous pipe is a kind of half-duplex communication, so a process can just read or write from pipe. In fact, reading and writing data to a pipe file is actually reading and writing to a kernel buffer.

2.2 Usage of pipe

At first, we declare a array mypipe[2] and using pipe(mypipe) to create a pipe, where pipe[1] is the write side and pipe[0] is the read side.

After we generate a new child process using fork(), it will share the pipe with its father. Because pipe is half-duplex communication, so the process responsible for writing needs to close mypipe[0] and process responsible for reading needs to close mypipe[1].Then they can use I/O functions such as read(), write() to communicate.

Finally, the writing process close mypipe[1], this behavior will send a EOF to pipe, after read the EOF, the reading process can close mypipe[0] to end the communication.

2.3 Implementation

In my lab program, I let parent process read from source file and write to pipe, while child process read from pipe and write to destination file. And I set variable BUFFER_SIZE as the size of buffer.

At first, the program checks the number of arguments. If argc is not equal to 4, it means that the user doesn't run the program in correct way: ./Copy <SrcFile> <DestFile> <BufferSize>. The program will print error message and exit.

The the program create a child process, the core code of parent process is below. Unimportant parts such as error check are ignored.

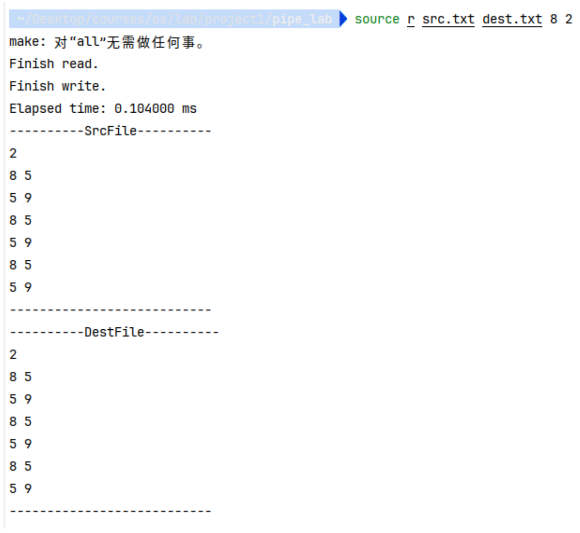
```c
FILE *src = fopen(argv[1], "r");
char *writeBuf = (char *) malloc(BUFFER_SIZE * sizeof(char));

//read from file and write to pipe
while(fread(writeBuf, sizeof(char), BUFFER_SIZE, src) > 0) {
    write(mypipe[1], writeBuf, BUFFER_SIZE);
    memset(writeBuf, 0, BUFFER_SIZE * sizeof(char));
}
```

Similarly, we also put the core part of child process here.

```c
//read from pipe and write to file
while(read(mypipe[0], readBuf, BUFFER_SIZE) > 0) {
    fwrite(readBuf, sizeof(char), BUFFER_SIZE, dest);
}
```

In this way we have achieved the main requirements of the lab. After finishing, we run the script and the output is below, which displays that the program runs correctly.



```
~/Desktop/courses/os/lab/project1/pipe_lab  source r src.txt dest.txt 8 2
make: 对“all”无需做任何事。
Finish read.
Finish write.
Elapsed time: 0.104000 ms
----------SrcFile----------
2
8 5
5 9
8 5
5 9
8 5
5 9
---------------------------
----------DestFile---------
2
8 5
5 9
8 5
5 9
8 5
5 9
---------------------------
```

Figure 1: The result of script "r".

2.4 Performance Analysis

We let the source file contains a 3n*n matrix. In the test, We test thirteen different sizes from 1 to 2^{12}. In each case, we repeated ten times to get a stable elapsed time. We use Gnuplot to visualize the test result, as shown in Figure 4.

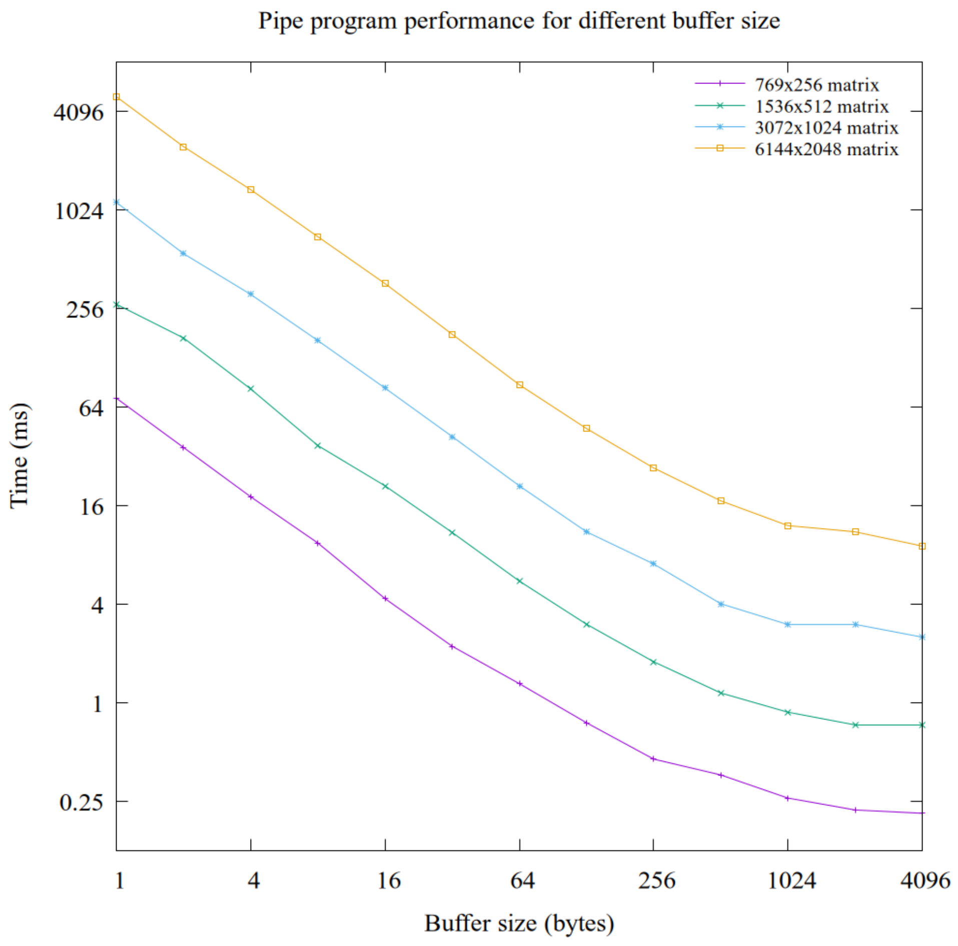


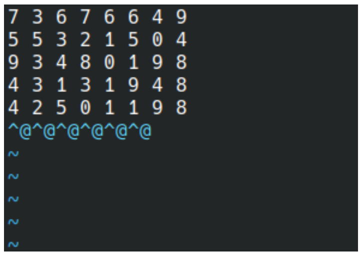
Figure 2: Performance for different input and buffer size.

From the Figure 2 we can find that when the buffer size is doubled, the program runtime is reduced by about half. However, this conclusion only holds when the buffer size is small, and we can see that when the buffer size is large, the runtime growth rate decreases and is more influenced by other factors, as the file copy only requires a few reads and writes.

2.5 Troubleshooting

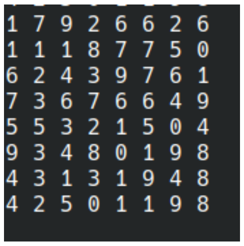
When checking the output of the program, I found that there were some strange signs at the end of file. As shown in Figure 3.

After searching on the Internet, I known that the sign is a visible format of '\0'. The reason for this symbol is that when write the data into file through buffer, it's hard for data to fill the buffer just right, so the zeros in buffer will be output to file with data. This makes the signs in the file. Consider the principle of program, the signs are inevitable. However, these signs have less impact on the subsequent operation of the file. We can see in Figure 4 that the content of file is displayed correctly when using cat command.



```
7 3 6 7 6 6 4 9
5 5 3 2 1 5 0 4
9 3 4 8 0 1 9 8
4 3 1 3 1 9 4 8
4 2 5 0 1 1 9 8
^@^@^@^@^@^@
~
~
~
~
~
```

Figure 3: Strange signs at the end of output.



```
1 7 9 2 6 6 2 6
1 1 1 8 7 7 5 0
6 2 4 3 9 7 6 1
7 3 6 7 6 6 4 9
5 5 3 2 1 5 0 4
9 3 4 8 0 1 9 8
4 3 1 3 1 9 4 8
4 2 5 0 1 1 9 8
```

Figure 4: The output of cat command.

3 Shell Lab

3.1 Socket programming

In this lab, I created a socket-based server. The program uses socket() system call to create a socket. Then it calls bind() to bind the socket with address and port. Note that ports 0 through 1023 are reserved ports, and binding these ports will cause ”Permission denied” error. Finally, it uses listen() system call to start listening for connections. It will connect to client using accept() system call. At this point the program will block until a client tries to connect, which affects the shutdown of the server and we will talk about it later. After all things down, close() is run and program exits.

On the client side, things are easier,it can just create a socket and connect to server. In this lab, we use ”telnet” tool to do these things for us.

3.2 Main structure

After connecting to a client, the program will create a child process and the child process will handle the requests of client. Meanwhile, the father continue waiting for a new client. The main logic is below.

```
//init
//create socket(socketfd)
//bind
//listen
while(1) {
    //accept, create new socket(newsocketfd)
    //other things, talked about later
    //fork
    if(pid == 0) {
        //child handle requests
    } else {
        //close newsocketfd to let child process work.
    }
}
//close socketfd
```

3.3 Command execution

As talked before, the child process handle the requests of client. It can communicate with client by read newsocketfd. After getting the request, it calls parseLine() function to split input to argv[] and forwards it to shell_exec() to run the command. shell_exec() calls different functions according to argv[1].

3.4 Pipe symbol

To implement pipe symbol, the program traverses the argv[] and record the number of pipe symbols. If there's no symbol, the command will be executed simply. Else a pipe array declared as pipefd[pipe_count][2] will be created. And the child process creates the same number of grand processes as the number of commands. Grand processes use dup2() to redirect the input and output, and communicate by pipes. Finally the output of whole command will be print on client's terminal.

3.5 Graceful shutdown

To shutdown a server, I used self-build function kbhit() to check the input non-blockingly. After reading letter 'q' from input, the server will come in exit program. As talked before, the program will be blocked at accept(), so the server must begin closing officially after a client tries to connect :(. At this point, the program will wait all clients disconnects and do cleaning then exit.

4 Matrix Lab

4.1 Matrix multiplication algorithms

We denote the multiplication as $C = A \cdot B$. To make question easier, we set the size of matrix is power of 2.

The first algorithm is obvious, just traverse every point of C and compute it using

$$C[row][col] = \sum_{i=0}^{Size} A[row][i] \cdot B[i][col]$$

The second algorithm divides the matrices into 4 parts:

$$\begin{aligned} C_{11} &= A_{11} \cdot B_{11} + A_{12} \cdot B_{21} \\ C_{12} &= A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\ C_{21} &= A_{21} \cdot B_{11} + A_{22} \cdot B_{21} \\ C_{22} &= A_{21} \cdot B_{12} + A_{22} \cdot B_{22} \end{aligned}$$

The last algorithm called Strassen's Multiplication:

$$\begin{aligned} P &= (A_{11} + A_{22}) \cdot (B_{11} + B_{22}) \\ Q &= B_{11} \cdot (A_{21} + A_{22}) \\ R &= A_{11} \cdot (B_{12} - B_{22}) \\ S &= A_{22} \cdot (B_{21} - B_{11}) \\ T &= B_{22} \cdot (A_{11} + A_{12}) \\ U &= (A_{21} - A_{11}) \cdot (B_{11} + B_{12}) \\ V &= (B_{21} - B_{22}) \cdot (A_{12} + A_{22}) \end{aligned}$$

$$\begin{aligned} C_{11} &= P + S - T + V \\ C_{12} &= R + T \end{aligned}$$

$$C_{13} = Q + S$$

$$C_{14} = P + R - Q + U$$

According to earlier tests, the running time of algorithms 2 and 3 is four times that of algorithm 1. Consider the complexity of memory allocation. We use algorithm1 in the lab.

4.2 Thread pool

In the program, I have tried two types of multi-threaded calculations. One is just use pthread library simply, another is using a thread pool. Thread pool creates a task queue, threads automatically take the tasks and run them. It's a efficient way to manage threads.

We use some main functions to complete the computation, pool_create() initializes a thread pool. push_task() pushes tasks into queue for threads to take. pool_wait() blocks the program until there's no task and working thread. pool_destroy() clean the thread pool and free the memory.

4.3 Performance analysis

We let the source file contains two n*n matrix. In the test, We test 8 different sizes from 2^5 to 2^{12} and 6 different number of threads from 1 to 2^5. In each case, we repeated ten times to get a stable elapsed time. We use Gnuplot to visualize the test result. The result is shown in Figure 6.

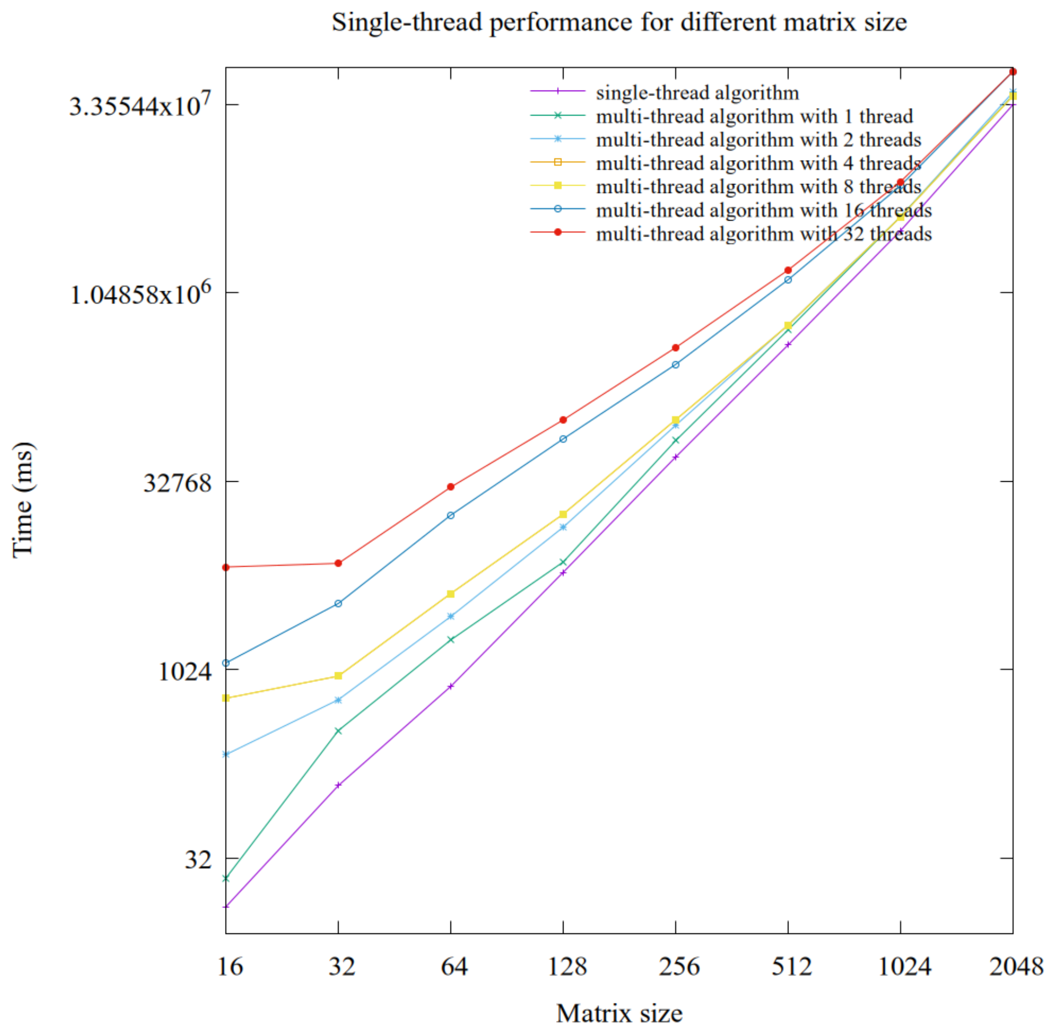


Figure 5: Performance for different size matrices and number of threads.

In the figure you can find that the single-thread algorithm is the most fast. This is because for multi-thread algorithm, it takes time for processes to take tasks, synchronize and communicate, which leads to poor performance of multi-thread algorithm. You can also see that the gap between multi-thread and single-thread algorithms decreases as the matrix size gets larger, because the computational benefits of multi-thread compensate to some extent for the process synchronization slowdowns associated with multi-thread.

It is worth noting that with 16 and 32 threads, even when computing large matrices, there is a significant performance difference. I think this is because the number of threads exceeds the number of physical cores on my computer, so that each thread cannot really run in different cores at the same time, with some performance loss from the scheduling.