Project 1 Report

王智烨 521021911051

Project Files

The complete list of project files are listed below

```
Prj1+521021911051
    report.pdf              #The PDF version of the report
    report.md               #The MD versionof the report
    Typescript.pdf          #Displaying the input and output of each program
    GNUplot                 #used for GNUplot, can be used to check for origin
data
        Copy.dat
        Copy.plt
        Matrix.dat
        Matrix.plt
    Matrix_test             #Matrix sample of different size, used for test
        Matrix3.in
        Martix10.in
        Matrix50.in
        Matrix100.in
        Matrix250.in
        Matrix500.in
        Matrix1000.in
    makefile                #makefile
    Copy.c                  #Copy is implemented here
    shell.c                 #shell is implemented here
    multi.c                 #multi is implemented here
    single.c                #single is implemented here
    src.txt                 #Source of the Copy
    dest.txt                #Output of the Copy
    data.in                 #Input of the single and multi
    data.out                #Output of the single and multi
    random.out              #Output of the multi in the random mode
```

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 one to another. The way it is implemented is to use 2 processes. One is responsible for reading and the other writes. These two processes transfer the content through a pipe, and the user can set the size of the buffer, which will affect the speed of copy.

- Command line

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

Analysis

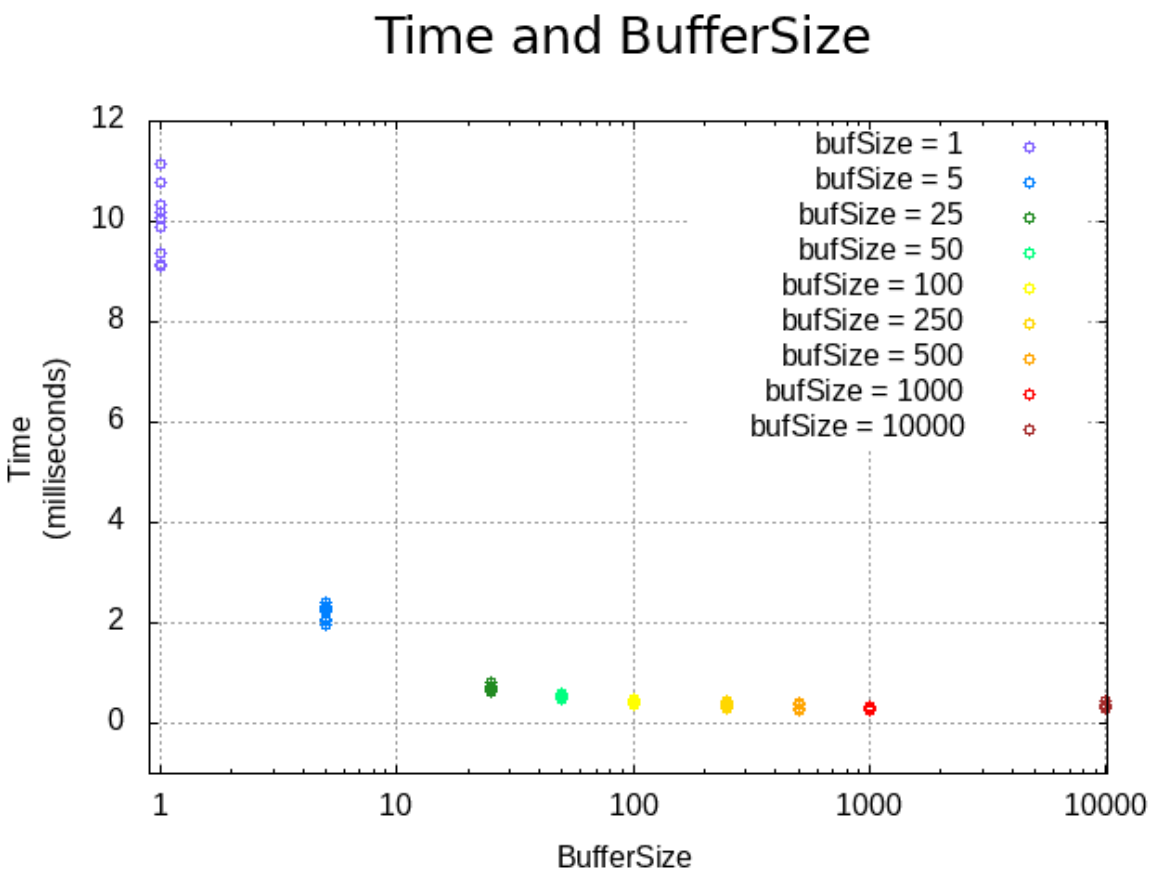


Figure 1 Time and BufferSize

In my tests, I used the sample_data.in file, which, according to the program output, was 43849 bytes in size. In the experiment, the pipe with the BufferSize of 1, 5, 25, 50, 100, 250, 500, 1000 and 10000 bytes was copied and timed respectively. The timing results were shown in *Figure 1 Time and BufferSize*, the X-axis of which is logarithmic.

It's worth noting that as BufferSize goes up, the copy time doesn't linearly decline. When BufferSize is greater than 5, the increase in Buffer does not significantly reduce the replication time. My idea about this phenomenon is that when buffersize is only 1, buffersize is indeed the main factor affecting the copy speed, because we can find that when buffersize is increased to 5 times of the original, the running time is also nearly shortened by 5 times. But from this point on, buffersize is no longer the main factor affecting the running speed of the program, so even if buffersize has been increased to a hundred times, a thousand times or even ten thousand times, there is no significant change in the running speed.

Part 2: Shell

Introduction

The function of the shell section is to write a shell-like program as a server, which should handle the commands with arguments and the commands connected by pipes. Also, the program should use multi processing to support more than one client.

- Command line

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

- Command line for telnet

```
telnet localhost <Port>
```

Analysis

In this part, I use `fork()` to generate sub-processes to manipulate the communication with the client. When the parent process accept a servant, it will fork a sub-process and wait for other potential contact.

When it comes to the sub-process, it will first handle the input of the client. First, it will divide the input strings according to spaces. While traversing each string, it checks if there is a pipe symbol and whether the use of the pipe symbol is legal. After this, it will begin to execute commands with the function `execvp()`;. Similarly, program will fork again to make its sub-processes execute the commands because `execvp()` will take over the process that calls it.

The situation gets a little more complicated when the pipe commands exist. The program should be capable to redirect the input and output of some sub-processes. For instance, when the server receives `ls -l | wc | wc`, it will fork 3 sub-processes first. And then, the first sub-process redirect its out put to the write end of the pipe between the second sub-process and itself before executing `ls-l`. For the second one, it redirect its input to the read end of the pipe between the first pipe and itself first, redirect its output to the write end of the pipe between the third pipe and itself and execute the `wc`.

Here is the sample code with comments:

```
for (int i = 1; i < *pipe_count + 2; ++i)
{
    if (pipe(fds)== -1)
    {
        perror("pipe create error");
        exit(2);
    }
    int ret = fork();
    if (ret < 0)
    {
        perror("pipe create error");
        exit(2);
    }
    else if (ret == 0)                                          // child
process
        {
            pid_pipe[i] = getpid();
```

```
            dup2(previous_out_fd, STDIN_FILENO);                    //redirect
input
            close(previous_out_fd);
            close(fds[0]);
            if (i != *pipe_count + 1)                               //not the
last sub-process
                dup2(fds[1], STDOUT_FILENO);                        //redirect
output
            else //the last sub-process
            {
                close(output[0]);
                dup2(output[1], STDOUT_FILENO);
            }
            close(fds[1]);
            if (execvp(pipe_commands[i - 1][0], pipe_commands[i - 1]) == -1)
            {
                perror("execvp");
                exit(2);
            }
            exit(0);
        }
    else                                                            //parent
process
    {
        previous_out_fd = fds[0];
        close(fds[1]);
    }
}
```

Some thoughts and findings

When I first need to use `fork()` to create more than one sub process, I just simply put `fork()` into a for loop.

```
for (int i = 0; i < 3; ++i)
{
    fork();
}
```

However, this behavior will leads to a terrible consequence which is that after the first loop, I have 2 process executing the for loop, both of them fork again. So after the for loop, I will have more than four sub-processes.

The right way is to use `getpid()` to check whether the process is a sub-process or a parent process. And we can also maintain a pid_array to record the pid of each sub-process.

```
int pid_array[4];
pid_array[0] = getpid();
for (int i = 0; i < 3; ++i)
{
    int tmp = fork();
    if (tmp == 0)
    {
        pid_array[i + 1] = getpid();
```

```
            break;
        }
        else if (tmp > 0)
            pid_array[i + 1] = tmp;
        else
        {
            perror("ERROR:fork");
            exit(2);
        }
    }
```

Part 3: Matrix multiplication

Introduction

In this section, I have written 2 programs to do matrix multiplication. One is single thread, doing the multiplication in the normal way. The other is a multithread program, using the principle of block matrix. Each thread is responsible for the multiplication of a block matrix, and finally the result of each thread is combined to a complete matrix.

- 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"
```

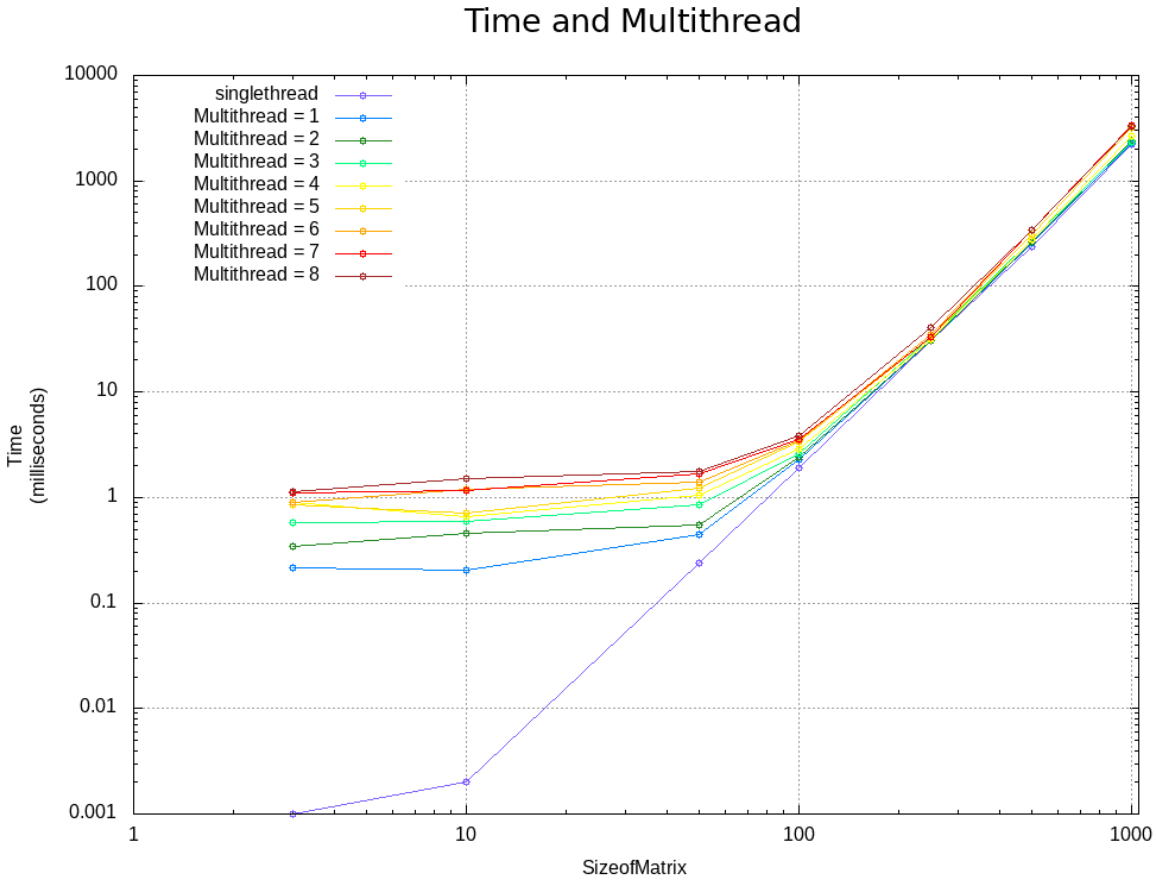
Analysis

In this experiment, I allocated 6 cores to my linux virtual machine in VMwhare.

```
top - 20:06:21 up 36 min,  1 user,  load average: 0.00, 0.05, 0.13
Tasks: 316 total,   1 running, 315 sleeping,   0 stopped,   0 zombie
%Cpu0  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu1  :  0.3 us,  0.3 sy,  0.0 ni, 99.3 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu2  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu3  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu4  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu5  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :  12061.8 total,   9561.6 free,   1228.7 used,   1271.5 buff/cache
MiB Swap:   2048.0 total,   2048.0 free,      0.0 used.  10557.2 avail Mem
```

Figure2 Cores allocated

To test the speed of multithreading, I generated seven random matrices of different sizes, 3x3, 10x10, 50x50, 100x10, 250x250, 500x500 and 1000x1000, where each element is no larger than 100. I first used a single-threaded program to perform the matrix operations, followed by a multithreaded program to I first use a single-threaded program to perform the matrix operations, followed by a multi-threaded program to perform the "single-threaded" calculations. Considering that only one thread is computing, but a thread is waiting to receive the return value from the other thread, it should be two threads to be exact. Then 2 to 8 threads are used for the matrix calculation. The test results are shown in the *Figure3 Time and Multithread*.



Graph3 Time and Multithread

At the first sight, the result may be a little confusing. The program implemented multithread calculate the matrix multiplication slower. But we can find that when the program do the 3X3 matrix multiplication, the multithread using one thread to calculate takes much longer time than the singlethread. Based on this phenomenon, I guess it is because it takes a lot of time to generate threads and retrieve them, which is much more than the time used to compute there matrices.

Another question is whether multi-threading is useful here or not? I think there is still some usefulness. According to the curve on the *Figure3 Time and Multithread*, as the size of the matrix grows, the growth rate of the time consumed by multi-threaded operations is slightly smaller than that of single threads. Therefore, we can guess that the advantage of multithreading may come into play when the problem size rises further.

Moreover, when the program allocates more threads than the number of cores, the time for computing does not increase significantly. My guess is that when the number of threads increases, although some of the threads need to wait for other threads to finish their computations before they can start, they take up less time because the amount of computations needed per thread is reduced at this time, so no significant increase in time consumption can be observed.