

Project 1 Report

花振东 jackeyhua@sjtu.edu.cn

2024 年 3 月 30 日

1 copy

1.1 Description

First, we need to write a copy file using C API `read` and `write`. The program should be able to copy a file from one location to another. I use the buffer size of 100.

Then, we need to use `fork` to create a child process and use `execl` to call the `MyCopy` program to copy the file. The parent process will wait for the child to finish.

Last, we need to use `pipe` to realize file copying. We first fork a child process, and then in the child process, we fork another one. The child will open the file and write to pipe, and the grandchild will read from the pipe, and write them to the destination file.

1.2 Usage

To compile the sub project, we can change the directory to `proj1/copy` and run `make MACRO=TIME1, TIME2`.

To run the program, we can use `./copy <srcpath> <dstpath>` to copy the file from `src` to `dst`.

More details, please refer to the type script of Project 1.

1.3 Trick

I use conditional compilation to control the output of the time. I use `TIME1` macro to signal the use of `clock()` to count time. I use `TIME2` macro to signal the use of `gettimeofday()` to count time. And then i use `#ifdef TIME1` and `#elif TIME2` to control the output of the time.

That is why we need to state the macro when compiling via `make`.

1.4 Result

I use `copy/test.py` to run automatic tests. It will compare the three copy methods on same configuration, namely macro and size of source file. Then, it will change the size of the source file and check the trend of the time.

The result is plotted by matplotlib and saved in `copy/plot1.png` for TIME1 and `copy/plot2.png` for TIME2. Check Figurec1 and Figure 2 for the result.

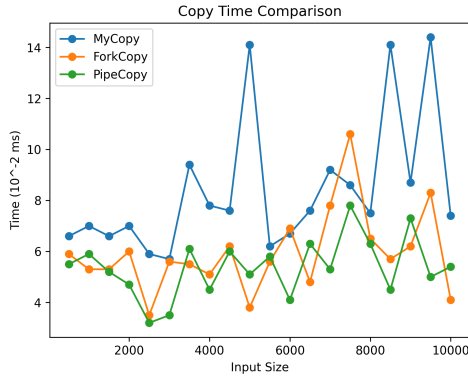


图 1: Time of copy with `clock()`,
buffer size = 100

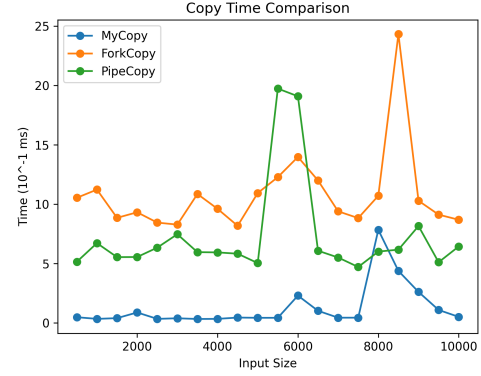


图 2: Time of copy with
`gettimeofday()`, buffer size = 100

Then, i perform the test again under buffer size = 1024 and 2048. Check Figure ?? and Figure 4 for the result of buffer size = 1024. Check Figure 5 and Figure 6 for the result of buffer size = 2048.

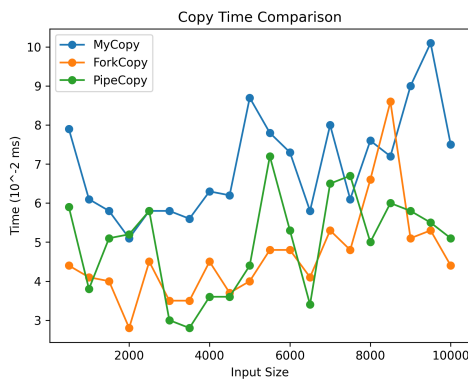


图 3: Time of copy with `clock()`,
buffer size = 1024

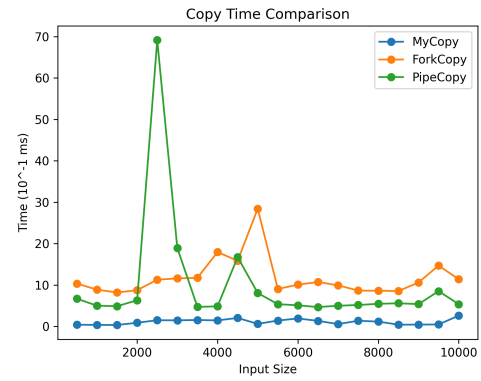


图 4: Time of copy with
`gettimeofday()`, buffer size = 1024

As is depicted above, we can see that the time recorded by `clock()` fluctuates a lot, while the time recorded by `gettimeofday()` is comparatively more stable. This is

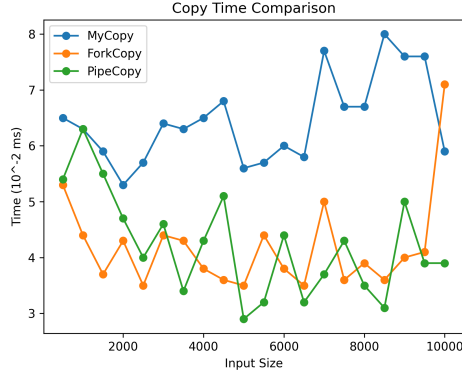


图 5: Time of copy with `clock()`,
buffer size = 2048

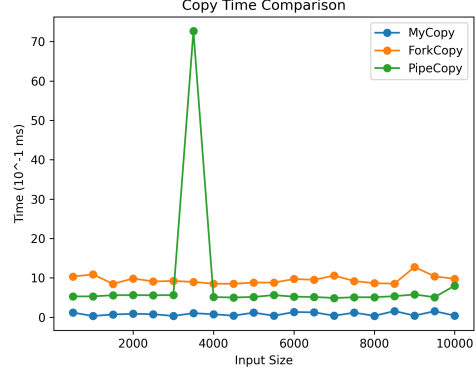


图 6: Time of copy with
`gettimeofday()`, buffer size = 2048

possibly due to the fact that `clock()` returns the processor time and `gettimeofday()` returns the system time. And i prefer to believe that the latter one is more precise.

We can see from the figures of `clock()` method that in general, MyCopy takes the longest time and the rest two have comparable performance. It is actually really wild, because ForkCopy use MyCopy to realize the function, but it is faster than MyCopy. I guess that is because the overhead of fork is not that large and that when testing ForkCopy, the src and dst file are in the cache, which makes the copy faster. It is also wild that PipeCopy is obviously faster than MyCopy, they nearly use the same way, both via read and write to a particular file descriptor. And the former one needs additional time to create a pipe. I guess that is because pipe has better performance than function calling in MyCopy and fork and pipe have been systematically optimized. We do not see regular and obvious change in time when buffer size changes.

Then, we focus on the result of `gettimeofday()` method, which is much more stable. First, we can obviously see that MyCopy has less time than PipeCopy, and PipeCopy has less time than ForkCopy. It corresponds to the analysis above. ForkCopy calls MyCopy, so it should be slower. PipeCopy needs time to establish a pipe, so it also should be slower than MyCopy. We can also see that time of copy will be more stable when buffer size grows. I think the reason is that it fully utilize the pages of OS. Typically the page is 4KB. 100 is not a factor of 4096, while 1024 and 2048 are. So in buffer size 100, we see great fluctuation, because it can not perfectly fill a page. But in buffer size 1024 and 2048, not only can it fill a page, but also can it hold more data, which makes the time more stable. We can also find that there always exists an outlier in PipeCopy, but not in MyCopy and ForkCopy. I guess that is possibly due to the mechanism of pipe and blocking I/O. When the pipe buffer is full, the writer may be blocked and put to

sleep. Same happens to the readers when the buffer is empty. Then wake up the writer or reader will take additional time and context switch. That is why we see the outlier in PipeCopy.

2 shell

There is no performance test in the sub project, so i will just share some of the tricks.

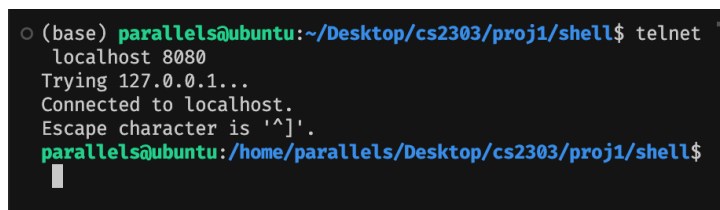
2.1 Description

We need to write a program that mimic Linux shell, supports some of frequently used commands, such as `ls`, `pwd`, and then make it a server and client program. The server will receive the command from client and execute it, then send the result back to the client.

2.2 Usage

To compile the sub project, we can change the directory to `proj1/shell` and run `make`.

To run the program, we can use `./server <port>` to run the server program and `telnet localhost <port>` to connect to the server. After connection, a shell that really looks like bash will appear, check Figure 7 for the screenshot. I choose the color that is same to the bash. You can differentiate the program with bash by the path, if the path starts with `~`, then it is bash, otherwise it is the shell program. Then, you can type any commands and the server will execute it and send the result back to the client.



```
○ (base) parallels@ubuntu:~/Desktop/cs2303/proj1/shell$ telnet
localhost 8080
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
parallels@ubuntu:/home/parallels/Desktop/cs2303/proj1/shell$
```

图 7: Screenshot of shell

2.3 Trick

The pipeline of this shell is that, we first read the input from the client, then parse the input into separate commands, and check whether it is `exit` or `cd`. If it is, then we will exit the program or change the directory. Otherwise, the parameters will be passed

to my `exec` function. It will check whether there is a pipe in the command. If not, it will call `execvp` or `execv` based on whether the command is a self supported one (Typically, `ls`, `pwd`, `rm`, `cat`, `wc`). If so, it will call my version of implementation, otherwise it will directly use the Linux system call for robustness. It is worth noting that we need to set the standard output and standard error to be client file descriptor, so that the result can be sent back to the client. If there is a pipe, it will fork another child process that deal with the left part of pipe, and the parent process will recursively call `exec` to deal with the right part of pipe. They will communicate via pipe and file descriptor redirection.

In order to reuse the port, i set the `SO_REUSEADDR` option to the socket.

There are one main bugs that puzzles me, which is worth learning. The carriage return in Linux is `\n\r`. It is **not** `\n`. It leads to the error in arguments parsing, and can not see any differences when printing the parsed argument array.

2.4 Reflection

Shell is nothing more than a program running in infinite loop, parsing user input and calling corresponding executable to deal with. This sub project makes me realize that we can easily design a personalized shell.

The design of file descriptor is also brilliant. It masks the source or destination of operation. So it gains generality.

3 sort

3.1 Description

We need to write merge sort both in single thread and multi thread (using PThreads API).

3.2 Usage

To compile the sub project, we can change the directory to `proj1/sort` and run `make`.

To run the program, we can use `./MergesortSingle < <input data> > <output data>` and `./MergesortMulti < <input data> > <output data>`. Here, we use redirection via `<` and `>`, because we rely on `scanf` to read the size of array and the elements of array.

3.3 Result

I use `sort/script.py` to run automatic tests. It will compare the two methods on same configuration, namely size of array. Then, it will change the size of the array and check the trend of the time.

The result is plotted by matplotlib and saved in `sort/plot.png`. Check Figure 8 for the result.

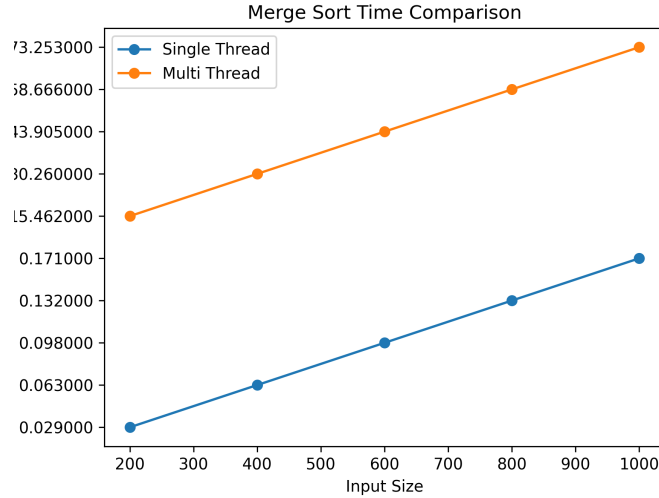


图 8: Time of sort

We can see two obvious trend. The time of sort grows as the size of array increases, which is intuitive. And the time of multi thread is much greater than the single thread. Before analysis, I need to state how this program utilize multi process. When given an array, it will divide the array into two parts in the middle, and then create two threads to sort the two parts. Then it will recursively call itself until the array has size 1. I guess there are two main reasons that multi thread is slower than single thread. **The first reason** is that the parallel desktop of my Macbook Pro only has 2 process, namely ``nproc` = 2`, but the array size is very big, which may use more than 10 threads (corresponding to size 1024). So the system need to switch the context frequently, which will take additional time. **The second reason** is that the creation and join of thread will take additional time. And the system need to allocate more memory for the threads, which will also take time.

And I guess there is possibly another reason. Merge sort here is not a typical multi thread program. By typical, I mean that we can assign the number of thread in advance and divide tasks fairly to each thread, E.g. matrix multiplication. But here, the number of thread is actually dynamic. In every round of recursion, we assign two threads to sort the two parts. And given that the number of thread supported by the VM is only 2,

there is a huge number of context switch, so the early thread has to wait for a long time for the later thread to finish.

I do another test to check whether the first or the second reason has more influence. I change the size of array to $[2, 10]$ with gap size 2. This is corresponding to the number of thread supported by my virtual machine. The result is plotted in Figure 9. We can see that still again the time of multi thread is greater than the single thread. So i think the second reason has more influence.

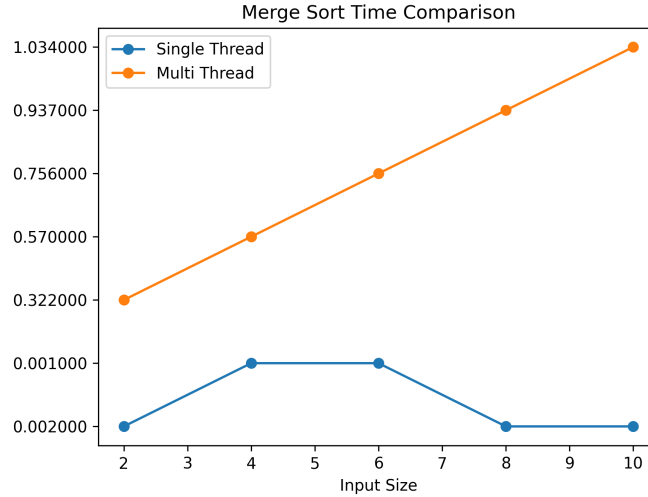


图 9: Time of sort with size $[2, 10]$

So practically speaking, we should avoid using more multi thread than the number of process supported by the system. And we should also avoid using multi thread when the size of array is too small because the overhead of creation and join of multi thread is huge.