Project 1 Report
Name: Fanyue Zhu Student id: 522031910547

1 Copy

1.1 Mycopy

1.1.1 Objectives

Write a C program that makes a new copy of an existing file using system calls for file manipulation.

1.1.2 Implementation

Open source file as read-only and destination file as read-write. The copy operation is implemented using the function `fread()` and `fwrite()`. Buffer size is set as 131702. The reason for this can be found in the analyze below.

1.2 ForkCopy

1.2.1 Objectives

Write a C program that creates a new process to copy the files using MyCopy. This program should spawn a new process using `fork` system call, and then use `execl` to execute MyCopy program.

1.2.2 Implementation

Fork a chid process using `fork`. In the child process, whose pid=0, transmit parameters and execute MyCopy program using `execl`. In the parent process, whose pid≠0, wait the child process using `wait(NULL)` .

1.3 PipeCopy

1.3.1 Objectives

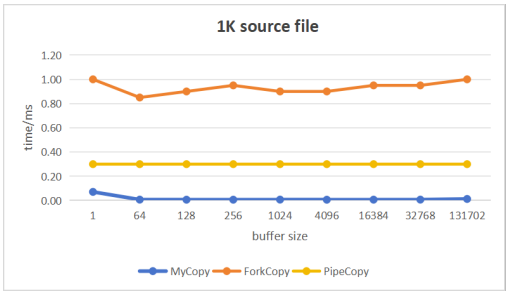
Write a C program to copy file that forks two separate processes: One for reading from the source file and the other for writing into the destination file. These two processes communicate using `pipe` system call.

1.3.2 Implementation

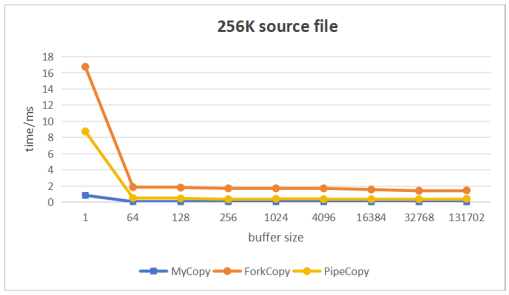
Create pipes.Then fork a process and set the child process as the writer while the parent process as the reader.Close the pipe that don't need.
In the child process, the writer use `fread()` to read from source file and `write()` to put it to the pipe's write end. And in the parent process, the reader use `read()` to get context from the pipe's read end and `fwrite()` to put it in destination file.
Remind that in the process `wait(NULL)` should be after the read operation. Otherwise, the data in pipe might be cleared after the child process end, leading to dead cycle.

1.4 Analysis

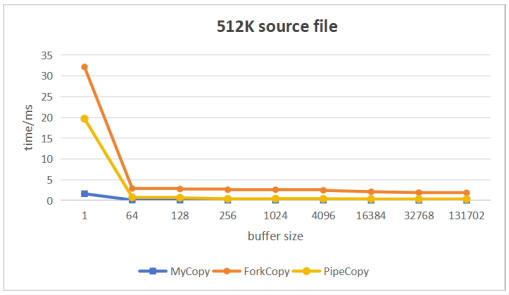
To compare the three copy methods, I test its execution time under various length of source file and buffer size.Each data is averaged over 20 runs.The test cases are as listed.
Source file size: 1K 256K 512K 1M 4M 16M
Buffer size: 1 64 128 256 1024 4096 16384 32768 131702
And the result are as follows.



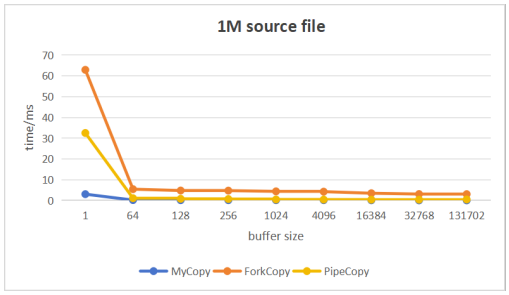
(a) Source file: 1K



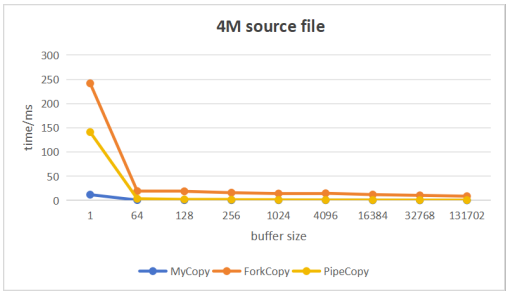
(b) Source file: 256K



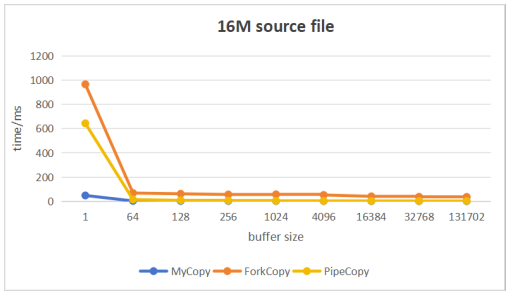
(c) Source file: 512K



(d) Source file: 1M



(e) Source file: 4M



(f) Source file: 16M

Figure 1: Execution time of three programs under different buffer size and length of source file

From the figure we can see that as the buffer size grows, the execution time of all reduces. At first, it reduces sharply. But when the size up to a certain value, often 64, the descent has levelled off, Stable around a certain value, stable around a certain value.
And among the three, ForkCopy costs the most execution time for all test file length, and PipeCopy ranks second. Considering the two programs both fork child process, we can figure out that pipe communication is faster than system call `execl`, especially when the buffer size is small. But as buffer size grows, the differences among the methods diminish.

2 Shell

2.1 Objective

Write a shell-like program that illustrates how Linux spawns processes. It should handle the commands with arguments and the commands connected by pipes.

2.2 Implementation

1. connect: Create and bind a socket, then listen and accept connection.
2. command phrase: First clear the at the end of the string received. Then phrase it using `strtok_r()`, which is thread-secure. If the pipe symbol `|` is get, then store its position in a array and increase the amount of pipes.
3. execute: Use the command `exevp()` and perror if error occurs.
4. command `cd`: Use the command `chdir()` to change the directory and `getpwd()` to get the location now. To realise `cd -`, there should be a variable store the old path. d
5. command `exit`: Break the dead-end loop of shell.
6. command pipe `|`: Use a loop to rephrase the command with `|` and redirect I/O from std to pipes using `dup2()`. The last command 's output should be redirected to socket.
7. zombie process prevention: Use sigaction to reclaim the resources of child process after it exits.
8. multiple client:Create a layer of dead-end loops at the outermost level and fork a child process to handle it whenever a new client is accessed. Every client quits with the command `exit`.

3 Mergesort

3.1 Objectives

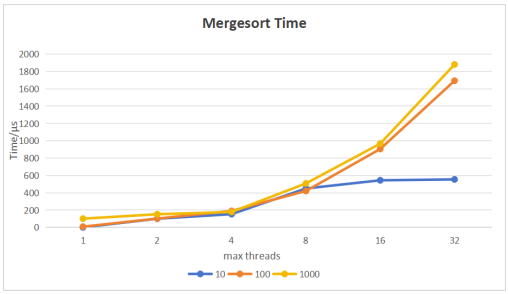
Implement mergesort using a single-threaded program and then with multiple threads (one of each divided component) and compare the times of the two implementations.

3.2 Implementation

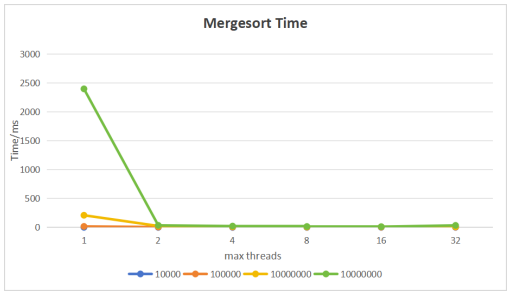
In the multi-mergesort, for every division the program will create a thread to do the right half part mergesort, while itself doing the left half part. If the level reaches the maximum, it won't create threads, and will do both part itself.

3.3 Analyze

From the figure we can see that when the data size is small, increase in max threads causes more runtime to be cost instead. We can figure out that the efficiency gains from concurrency don't have as much of an impact on latency as creating more threads when data sizes are small.
As the data size grows larger, the efficiency gained from concurrency begin to show its influence.



(a) data size:1e1-1e3



(b) data size:1e4-1e7

Figure 2: Execution time of mergesort under different data size and max threads