2020 OS Project 2 - Synchronous Virtual Device

Team Info

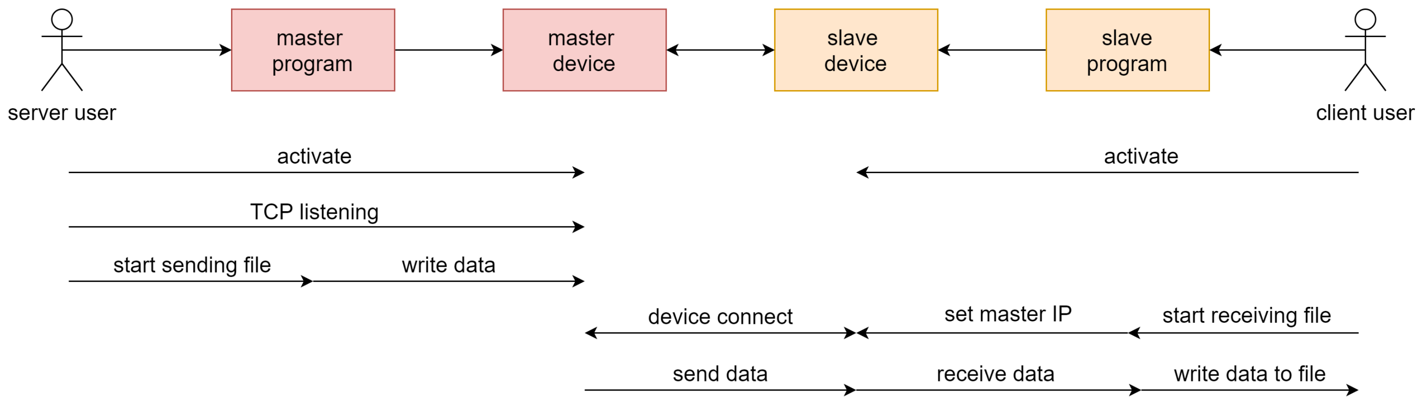
NTUCool Team Number : 51
Github repository : https://github.com/b07902067/OS-2020-Project2 (https://github.com/b07902067/OS-2020-Project2)
Team Member :
呂紹齊 B07902045
周敦翔 B07902050
郭宗穎 B07902067
蔡亞辰 B05611003
周俊廷 B07902091

Design

Kernel Version

4.14.25

Main idea



File I/O

Master.c

```
for each file
    while still have content (remain size[file] > 0)
        tranfer size = bufsize or (remain size[file] if < bufsize)
        read from file
        write to master
        remain size[file] -= transfer size;
```

After current input file is accessible, master will continuouly read the data and write to device until all data is transmitted. Repeat this process until all input files are completely processed.

Slave.c

```
        get file size info
        for each file
                cur_transfer_size = 0
                while filesize[file] > cur_transfer_size
                        read from master device
                        write to ouput file
                        cur_transfer_size+=read_size
```

Slave will continuously received the data transfer from the master, create new output file and write received data to correspond output files. Repeat this process until all files received are write to the output files.

Memory-Mapped I/O

master.c

The design is similar to `slave.c` , except that we need to write data to master device, thus an opposite workflow.

slave.c

Each time, we read 512 bytes from input file, which is stored in a char array `buf` and appended to `write_buf` , a place holding all the contents of single input file. Then `write_buf` is copied to `dst` , the virtual address where the memory mapping begins.

There are two different cases, depending on the size of input files. With respect to the first case, the size of input file is smaller than 100 times page size, namely, 409600 bytes, in which we can directly write the data into the memory-mapped files, whereas in the second case, the file is larger than the threshold. Thus we need to chop the data into many small chunks, then each instance can be treated like a small file in the first case.

master_device.c
In master_device.c add function to print in `dmesg` and memeory management such as `kfree` `kmalloc`

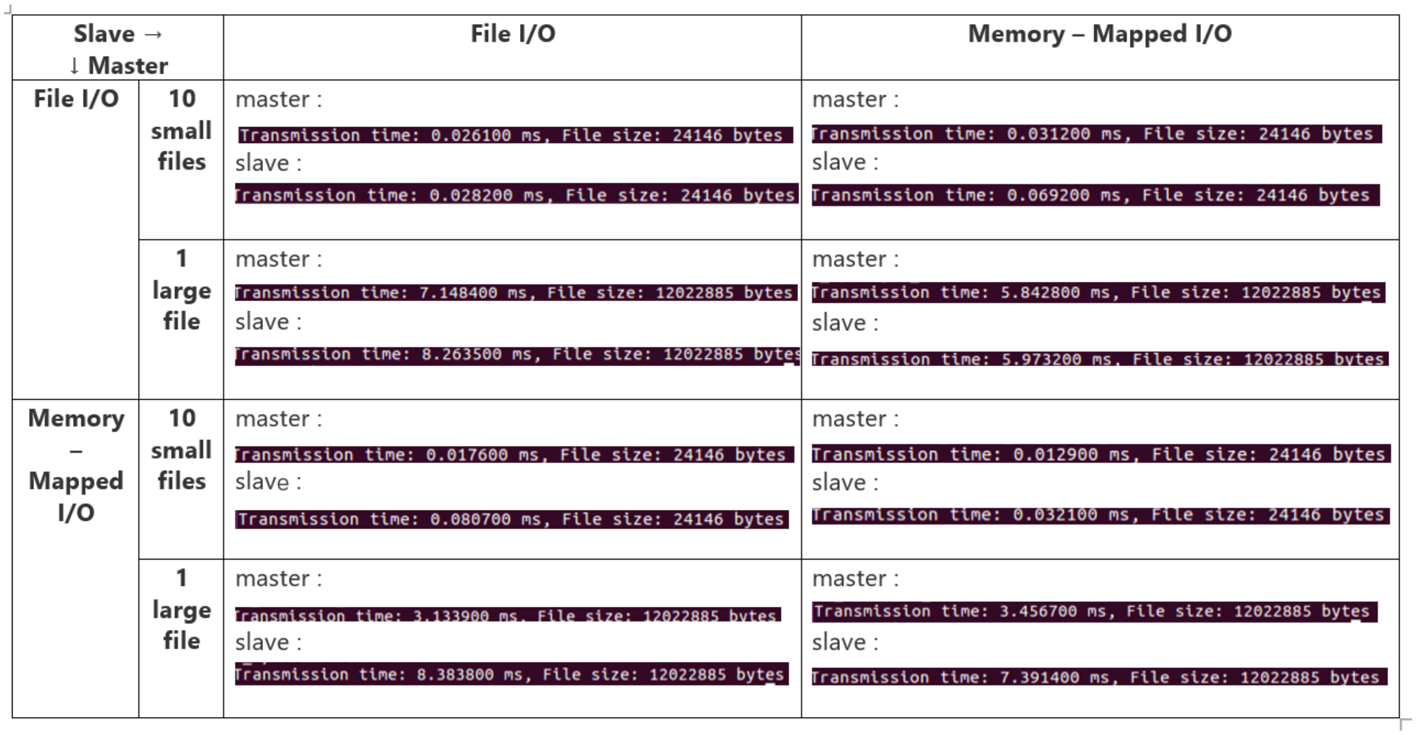
Problem - Unknown file size for files received from master

Description : Since the slave will continuously receive data transfer from the master, data from different files might transfer together without separation, result in the slave couldn’t decide where a file start and end.
Solution : Before the transfer begins, we first inform the slave about the size of all the files. Hence the slave could recognize where a file starts and termimntes based on file information.
psuedo code shown below:

```
int get_size(){
    int total = 0;
    char size_buf[20];
    for(int i = 0 ; i < N ; i++){
        struct stat st_buf;
        stat(filename[i], &st_buf);
        sprintf(size_buf , "%lld%c\0" , st_buf.st_size , special_char);
        int len = strlen(size_buf);
        if(method == 'f'){
            write(device_fd, size_buf, len);
        }
        else{
            write(sf_fd, size_buf, len);
        }
        total += len;
        total_file_size += st_buf.st_size;
        file_sz[i] = st_buf.st_size;
        }
    return total;
}
```

Comparison between File I/O and Memory-Mapped I/O

Sample Input Result



Slave → ↓ Master		File I/O	Memory – Mapped I/O

File I/O	10 small files	master :
		`Transmission time: 0.026100 ms, File size: 24146 bytes`
		slave :
		`Transmission time: 0.028200 ms, File size: 24146 bytes`
			master :
			`Transmission time: 0.031200 ms, File size: 24146 bytes`
			slave :
			`Transmission time: 0.069200 ms, File size: 24146 bytes`

	1 large file	master :
		`Transmission time: 7.148400 ms, File size: 12022885 bytes`
		slave :
		`Transmission time: 8.263500 ms, File size: 12022885 bytes`
			master :
			`Transmission time: 5.842800 ms, File size: 12022885 bytes`
			slave :
			`Transmission time: 5.973200 ms, File size: 12022885 bytes`

Memory – Mapped I/O	10 small files	master :
		`Transmission time: 0.017600 ms, File size: 24146 bytes`
		slave :
		`Transmission time: 0.080700 ms, File size: 24146 bytes`
			master :
			`Transmission time: 0.012900 ms, File size: 24146 bytes`
			slave :
			`Transmission time: 0.032100 ms, File size: 24146 bytes`

	1 large file	master :
		`Transmission time: 3.133900 ms, File size: 12022885 bytes`
		slave :
		`Transmission time: 8.383800 ms, File size: 12022885 bytes`
			master :
			`Transmission time: 3.456700 ms, File size: 12022885 bytes`
			slave :
			`Transmission time: 7.391400 ms, File size: 12022885 bytes`

Compare master performance

As shown in the table above,
The transmission time using **File I/O** on master :
0.026100 ms and **0.031200 ms** for **24146 bytes**
7.148400 ms and **5.842800 ms** for **12022885 bytes**

The transmission time using **Memory-Mapped I/O** on master :
0.017600 ms and **0.012900 ms** for **24146 bytes**
3.133900 ms and **3.456700 ms** for **12022885 bytes**

As we can observe,
File I/O might not be a bad choice to handle small files, but the performance of File I/O declines when the size of the file increases.

Memory-Mapped I/O has a shorter transmission time dealing with large files compared to File I/O. Besides, Memory-Mapped I/O also better transmission time while dealing with small files.

The master worked as a reader for input files. It reads the input files and then send to the slave for further procedures. Compared to reading files bytes by bytes into master program and then writing to the corresponding file descriptor (File I/O method), Memory-Mapped I/O, which allocates pages of data and copies buffer to target pages using `memcpy()` , saved more time. It does not need to read the data bytes by bytes into master program and write bytes by bytes to the corresponding file descriptor.

The `read()` and `write()` are slow and it is the key factor that makes the File I/O method having poor performance. The larger the file, the more `read()` and `write()` operations, the longer the transmission time. In terms of Memory-Mapped I/O, the whole pages of selected data are copied directly to target pages, thus faster compared to File I/O.

As we can see the difference between the transmission time of File I/O and Memory-Mapped I/O is not really that significant to indicate that which method is better. Though we are freed from copying memory directly using Memory-Mapped I/O, it also has several downsides. The setup cost for `mmap()` is actually quite high, considering TLB flushing must be executed to clean up the buffer after context switching. We must also take page faulting into account, which is an expensive operation. Briefly, the overhead of Memory-Mapped I/O is not necessarily lower than the cost of copying memory in a nice streaming manner.

Compare slave performance

As shown in the table above,
The transmission time using **File I/O** on slave :
0.028200 ms ***(File I/O)*** and **0.080700 ms** ***(Memory-Mapped I/O)*** for **24146 bytes**
8.263500 ms ***(File I/O)*** and **8.383800 ms** ***(Memory-Mapped I/O)*** for **12022885 bytes**

The transmission time using **Memory-Mapped I/O** on slave :
0.069200 ms ***(File I/O)*** and **0.032100 ms** ***(Memory-Mapped I/O)*** for **24146 bytes**
5.973200 ms ***(File I/O)*** and **7.391400 ms** ***(Memory-Mapped I/O)*** for **12022885 bytes**

Since the transmission time at slave might be affected by the method choose in master, we specified the method used in master in the bracket after the transmission time.

As we can observe,
When master use File I/O, Memory-Mapped I/O having better performance in larger files while File I/O have better performance in smaller file.

The reason for Memory-Mapped I/O in slave having poor performance in multiple small files when master use File I/O is because the data transfer to the slave is depends on the master. As the problem we stated in Design , when the file having smaller size than the page size, internal fragmentation occur. For each `mmap()` need to fill up the remaining page before `memcpy()` , which result in longer time for the data to fully copied to correspond output files. Besides, using File I/O will straight read and write to output files, which doesn't need to deal with internal fragmentation having better performance.

However, when come to larger file, as we stated before, the performance of File I/O decrease with the increase of file size. Because of more `read()` and `write()` used for larger file, File I/O result in poor performance. While for Memory-Mapped I/O, on the other hand having better performance. Memory-Mapped I/O didn't read the data into the program, but allocate the correspond page of the data locate and straight `memcpy()` to target output files. Since there is no need of having the data read into slave program, thus the transmission time of shorter.

When master use Memory-Mapped I/O, the Memory-Mapped I/O having better performance in both small files and large file.

Since the master is using memory mapped I/O, when deal with multiple small files, the slave will read the output files which have many blank space in the file. This is because the `memcpy()` in master will copied whole memory of page size.

When the file has smaller size than page size, the larger the blank. When slave using File I/O, slave might read many unnecessary blank thus waste many time.
When the file have bigger size, the File I/O method needs more `read()` and `write()` , which also result in poor performance.

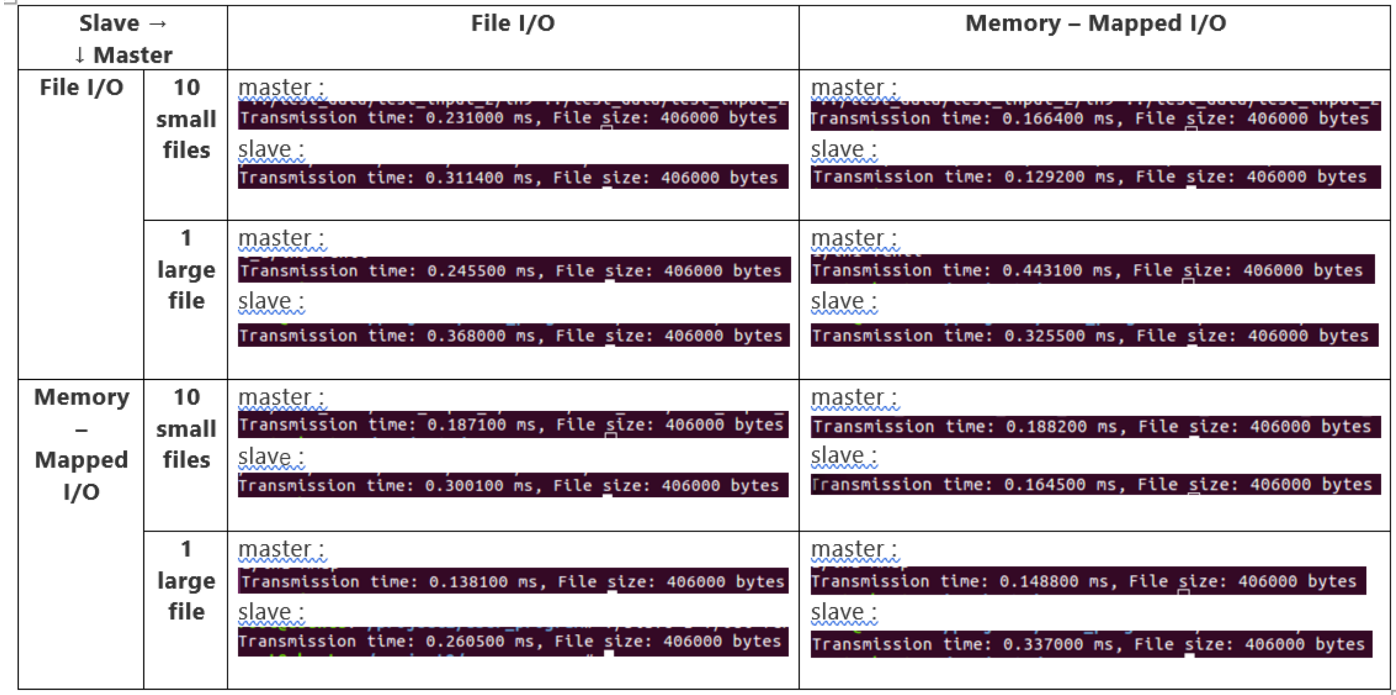
When Slave use Memory-Mapped I/O, it performance is better than File I/O in both multiple small files and large file. The reason is same as we stated before, Memory-Mapped I/O didn't have any data transfer in between the slave program, but straight `memcpy()` the pages with data to target output files.

Self Design Input Result

input - one large file with k file size and multiple files with total k file size

The self design input consists of two folder with same size, one contain a large file with 400k file size, and another contain 10 multiple files with sum up file size equal to 400k. We initially guess that if we implied Memory-Mapped I/O on both Master and Slave, the transmission on both cases will be same. We have such guess based on our design on Memory-Mapped I/O, which is the page need to be fully occupied before it can be transfer, which is the remaining blank need to be filled. Thus no matter there is only a large file or multiples small files, the transfer pages is always fully occupied. Since the page need to be fully occupied, no matter the page if completely or partially occupied by data, their transmission time is same.

Here is the result of our test:



Slave → ↓ Master		File I/O	Memory – Mapped I/O

File I/O	10 small files	master :
		`Transmission time: 0.231000 ms, File size: 406000 bytes`
		slave :
		`Transmission time: 0.311400 ms, File size: 406000 bytes`
			master :
			`Transmission time: 0.166400 ms, File size: 406000 bytes`
			slave :
			`Transmission time: 0.129200 ms, File size: 406000 bytes`

	1 large file	master :
		`Transmission time: 0.245500 ms, File size: 406000 bytes`
		slave :
		`Transmission time: 0.368000 ms, File size: 406000 bytes`
			master :
			`Transmission time: 0.443100 ms, File size: 406000 bytes`
			slave :
			`Transmission time: 0.325500 ms, File size: 406000 bytes`

Memory – Mapped I/O	10 small files	master :
		`Transmission time: 0.187100 ms, File size: 406000 bytes`
		slave :
		`Transmission time: 0.300100 ms, File size: 406000 bytes`
			master :
			`Transmission time: 0.188200 ms, File size: 406000 bytes`
			slave :
			`ransmission time: 0.164500 ms, File size: 406000 bytes`

	1 large file	master :
		`Transmission time: 0.138100 ms, File size: 406000 bytes`
		slave :
		`Transmission time: 0.260500 ms, File size: 406000 bytes`
			master :
			`Transmission time: 0.148800 ms, File size: 406000 bytes`
			slave :
			`Transmission time: 0.337000 ms, File size: 406000 bytes`

From the table above, we observed the result of Memory-Mapped I/O in master. Slave is not take in our discussion since the performance of slave might been affect by the performance of master.

The transmission time using **Memory-Mapped I/O** on master :
0.187000 ms and **0.188200 ms** for **multiple small files**
0.138100 ms and **0.148800 ms** for **one large files**

By the result, We can observed that the transmission time on both multiple files and a large file using Memory-Mapped I/O having a maximal different of 0.005 ms. Hence the hypothesis we stated before is not verified by our test.

For multiple files instance, there will be more procedures for open all files and gain their file descriptors. Beside, there also more `mmap()` since there are more files. When there is only one larger file, the page size we set is able to contain the whole file, hence there will be only one `mmap()` needed.

Conclusion about the performance and efficiency

Comparing transfer of files with two different file size, first has a large internal fragements while the other with smaller internal fragmentation. Both situation theoretically consume same pages size in total. We can clearly indicate that although the total file size are same but the performance is affected by the page size and number of file.

When we increase the file size, the closer file size compare to the page size, the better efficiency using memory mapping I/O. Memory mapped I/O will map complete multiples of page size. For multiple files with each file size is way too smaller compare to the page size, caused big internal fragmentation, result in filling the remaining blank of the page which is a waste for performance.

Teamwork Distribution

組員	學號	工作
呂紹齊	B07902045	Idea, Coding
郭宗頴	B07902067	Idea, Coding
周敦翔	B07902050	Idea, Report
蔡亞辰	B05611003	Idea, Report
周俊廷	B07902091	Idea, Report

References

https://man7.org/linux/man-pages/man2/mmap.2.html (https://man7.org/linux/man-pages/man2/mmap.2.html)
https://github.com/hbagdi/ksocket (https://github.com/hbagdi/ksocket)