

# OS Project2 Report

- [1. Design](#)
- [2. Results](#)
- [3. Explanation & Discussion](#)
- [4. Worklist](#)
- [5. References](#)

## 1. Design

The purpose of this project is to compare two ways of implementing I/O: the **mmap()** method, and the normal method using **fcntl()**.

The master and the slave connects to each other by their own device.

We redefine the mmap function in master\_fops and slave\_fops, and finish the case "MMAP" in ioctl. Note that the memory it used is in the kernel.

the slave device mmap:

```
case slave_IOCTL_MMAP:
    printk("krecv start");
    ret = krecv(sockfd_cli, file->private_data, MAP_SIZE, MSG_WAITALL);
    printk("krecv(%lu)",ret);
    break;
```

the master device mmap:

```
break;
case master_IOCTL_MMAP:
    ksend(sockfd_cli, file->private_data, ioctl_param, 0);
    printk("ksend!\n");
    break;
```

- master:

Because there may be more than one file, we have a loop to process each file.

The mmap method implementation is following.

The Master will send files to the slave. We split the file into segments and send the same size each time until all files are sent to ensure that we won't use excess memory in one send.

We use mmap to get a memory and then copy the segment from file to that memory. Then call the ioctl to process mmap in the device.

```

case 'm': //mmap: read()/write()
    send=0;
    while(send<file_size){
        now_send = (send+MAP_SIZE<=file_size)?MAP_SIZE:file_size-send;
        file_address = mmap (NULL, now_send, PROT_READ, MAP_SHARED, file_fd, send);
        kernel_address = mmap (NULL, now_send, PROT_READ|PROT_WRITE, MAP_SHARED, dev_fd, send);
        send+=now_send;
        memcpy(kernel_address, file_address, now_send);
        ioctl(dev_fd, 0x12345678 ,now_send);
        munmap(file_address, now_send);
        munmap(kernel_address, now_send);
    }
    //flush
    ioctl(dev_fd, 0 ,now_send);

```

- slave:

We use a loop to handle multiple files in slave, too.

The implementation of mmap method is shown below.

Compared with master, slave appends the file size first in order to access correct position of mmap.

Then mmap both segment of device and segment of file to memory, and copy the segment of file on memory to the segment of device.

Last, munmap both of them.

```

case 'm': //mmap : read()/write()
    ret=1; //to start run, set pos num
    file_size=0;
    while(ret>0){
        ret=0;
        ret = ioctl(dev_fd, 0x12345678);
        ftruncate(file_fd, file_size+ret);
        kernel_address = mmap (NULL, MAP_SIZE, PROT_READ , MAP_SHARED, dev_fd, 0);
        file_address = mmap (NULL, MAP_SIZE, PROT_READ|PROT_WRITE, MAP_SHARED, file_fd, file_size);
        memcpy(file_address, kernel_address, ret);
        file_size += ret;
        munmap(file_address, MAP_SIZE);
        munmap(kernel_address, MAP_SIZE);
    }
    //flush
    ioctl(dev_fd, 0, 0);
    break;

```

Bonus: We add asynchronous flags in each socket we build in ksocket function so the master can send the file asynchronously and the slave can receive the file asynchronously. The ksocket. So file should be compile again if we want to change the methods.

Transmit files asynchronously by typing: `sudo ./compile.sh ASYNC`

```
# in compile.sh
cd ./ksocket
make CFLAGS=-D"$1"
...
```







```
# in ksocket/Makefile
...
all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) CFLAGS=$(CFLAGS) modules
...
```







```
// in ksocket/ksocket.c
#ifdef ASYNC
    sk->flags |= FASYNC; // sk is a socket file
#endif
```

## 2. Results







To get the results below, we ran master first and then run slave.

### Master: mmap

 Files	 Property	 Master	 Slave	 Master 1	 Slave 1
<u>Untitled</u>	Size	Time	mmap time	Time	fcntl time
<u>File1</u>	1 bytes	1970.331000 ms	0.096000 ms	1968.296300 ms	0.053300 ms
<u>File2</u>	7 bytes	1968.124300 ms	0.238900 ms	1993.418200 ms	0.237900 ms
<u>File3</u>	16 bytes	2031.480200 ms	0.085900 ms	1967.502500 ms	0.083000 ms
<u>File4</u>	140 bytes	1973.027800 ms	0.511400 ms	1938.616200 ms	0.212600 ms
<u>File5</u>	1980 bytes	1942.759900 ms	0.079600 ms	1963.614800 ms	0.085900 ms
<u>File6</u>	16337bytes	2912.290800 ms	0.086900 ms	1986.176400 ms	0.308000 ms
<u>File7</u>	167809bytes	2003.758900 ms	0.343800 ms	2000.108100 ms	4.449000 ms
<u>File8</u>	1519743 bytes	2017.060100 ms	14.396000 ms	1963.781800 ms	22.904600 ms

 Files	 Property	 Master	 Slave	 Master 1	 Slave 1
<a href="#">File9</a>	3138410 bytes	1991.699600 ms	32.264300 ms	2914.656100 ms	25.956600 ms
<a href="#">File10</a>	6808098 bytes	2981.727500 ms	959.767000 ms	2996.004300 ms	944.665100 ms
<a href="#">File11</a>	17072267 bytes	3999.980900 ms	1036.994200 ms	2982.178900 ms	1953.477400 ms
<a href="#">File12</a>	208076800 bytes	24940.391300 ms	21986.339800 ms	23053.041700 ms	21066.653000 ms

#### Master: fcntl

 Master:fcntl	 Property	 Master	 Slave	 Master 1	 Slave 1
<a href="#">Untitled</a>	Size	<a href="#">Time</a>	Mmap time	Time	fcntl time
<a href="#">File1</a>	1 bytes	<a href="#">1047.653900 ms</a>	0.076100 ms	1084.359400 ms	0.086600 ms
<a href="#">File2</a>	7 bytes	<a href="#">1957.602900 ms</a>	0.275200 ms	1971.038600 ms	0.238300 ms
<a href="#">File3</a>	16 bytes	<a href="#">2052.570000 ms</a>	0.086800 ms	2997.124000 ms	0.057100 ms
<a href="#">File4</a>	140 bytes	<a href="#">2956.506600 ms</a>	0.078200 ms	1061.255600 ms	0.090600 ms
<a href="#">File5</a>	1980 bytes	<a href="#">1981.937900 ms</a>	0.138400 ms	2012.166200 ms	0.103700 ms
<a href="#">File6</a>	16337bytes	<a href="#">1961.893800 ms</a>	0.298700 ms	3019.924200 ms	1.112100 ms
<a href="#">File7</a>	167809bytes	<a href="#">2031.948000 ms</a>	4.723300 ms	1995.256800 ms	6.578000 ms
<a href="#">File8</a>	1519743 bytes	<a href="#">2950.677000 ms</a>	17.314900 ms	1047.358100 ms	17.383900 ms
<a href="#">File9</a>	3138410 bytes	<a href="#">2017.994000 ms</a>	16.441200 ms	2046.998600 ms	32.314700 ms
<a href="#">File10</a>	6808098 bytes	<a href="#">2010.828300 ms</a>	927.995500 ms	1999.497800 ms	42.374500 ms
<a href="#">File11</a>	17072267 bytes	<a href="#">3988.724000 ms</a>	1994.765400 ms	3051.813900 ms	1966.099200 ms
<a href="#">File12</a>	208076800 bytes	<a href="#">30033.617500 ms</a>	28064.222400 ms	27943.099100 ms	25943.653200 ms

The slave page descriptor of slave

```
[ 2317.198395] slave: F000FF53F000FF53
```

### 3. Explanation & Discussion

- There are total 12 test files. The smallest one's size is only 1 byte. The difference of the sizes between the largest file and the smallest is about  $10^9$ , while the difference of the size between each consecutive file is about twice to ten times.
- First, comparing the two different methods used by the slave when the master applies the same method, it can be seen that the gap between the two methods is not very large. In the case of smaller file sizes, the fcntl method has a better performance in time. Furthermore, when the file becomes larger, the transmission performance of mmap is relatively good and stable. On the other hand, when master applies the fcntl method and the file size becomes larger, slave with the mmap method is relatively stable.
- Secondly, when comparing the two tables above, it can be seen that the method adopted by the master has more important effect on the transmission than the choice of the slave. When using the mmap method, the overall performance is significantly better than using fcntl when the file size is relatively large. Though the fcntl method has an advantage when the file is relatively small, it only takes effect in a little degree which can be ignored.
- There will be such a data gap mainly related to the principle and operation process used by the two methods. In the general I/O file, the system checks whether there is data in the page cache. If there is no data, it copies the data to the buffer.
- mmap is a memory mapping function which maps the content of a file to a section of memory. By accessing this section of memory, the file can be read and modified. When using the mmap method, the system will save the corresponding page into memory. Therefore, when dealing with large files and accessing pages close to each others, page errors can be significantly avoided, so in high-speed file access.
- Compared with the general I/O method, it is usually necessary to put the data into the buffer first. Memory mapping eliminates the middle layer and speeds up the file access time. Hence, the mmap method will be more effective when the file is large. However, both methods have their advantages and disadvantages and applicable conditions.
- We can find that if data is extremely big, the mmap method will be slower than fcntl method. We think that the reason is that our remaining of memory is not big enough to allocate the data size. Therefore, system will reallocate the memory if there are some necessary process need to be executed. At the same time, the action of reallocate memory costs many time, so the time of transmitting big data based on mmap method will be more than based on fcntl method.

## 4. Worklist

- B06902032 楊則軒: debug and write the report, test different file types.
- B06505017 謝心默: debug, create the demo files, demo video, and write the report.
- B06705059 魏任擇: compile the kernel, some of experimental results explanation, write the report.
- B06902033 黃奕鈞: ksocket ASYNC implementation, write the report.
- B06902012 龔柏年: mmap coding implementation, for master and slave sending and receiving
- B05204033 吳哲安: find testing data and test, analyze them. write the report. make output files.

## 5. References

- <https://welkinchen.pixnet.net/blog/post/41312211-記憶體映射函數-mmap-的使用方法>