2020 OS Project 2 - Synchronous Virtual Device

Instructor: Prof. Chih-Wen Hsueh

1. Group Members & Contributions

- 劉正鴻 - D07944009

  Organize meeting, survey, write the shell script, report

- 徐豪 - D06944016

  Finish the codes including master.c and slave.c, survey, report

- 汪廷妍 - B04902062

  Master_device.c, run experiments, organizing results, survey, report

- 曾彬輝 - B05902099

  Find useful repositories, table result, survey, report

- 蕭恩慈 - B07902095

  Diff code, table result, report

- 林壯謙 - B07902092

  ?

2. ENVIRONMENT

Linux (Kernel version 4.14.25)

3. INPUT PARAMETER

./master [file num] [file directory] [method]

./slave [output num] [input directory] [method] 127.0.0.1

4. DESIGN

Added mmap function in master.c

```
int file_num = atoi(argv[1]);
char file_list[file_num][50];
for (int i=0; i<file_num; i++){
    strcpy(file_list[i], argv[i+2]);
}
strcpy(method, argv[file_num + 2]);
```

...

```
for (int i=0; i<file_num; i++){
     char file_name[50];
     strcpy(file_name, file_list[i]);
     offset = 0;
```

We use for-loop to record the number of files to apply it on multiple-files-function by setting the input files N, next refer to that amount and enter the data to be transferred, then set the transmission method either using fcntl or mmap.

For mmap function in master.c, we use this method below:

```
case 'm'://mmap
        kernel_address = mmap(NULL, PAGE_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, dev_fd, 0);
        while(offset < file_size){
                size_t tmp = PAGE_SIZE;
                if (tmp > (file_size - offset)) tmp = file_size - offset;
                file_address = mmap(NULL, tmp, PROT_READ, MAP_SHARED, file_fd, offset);

                page_dicrs[page_num] = file_address;
                page_num += 1;

                memcpy(kernel_address, file_address, tmp);
                ioctl(dev_fd, 0x12345678, tmp);

                offset += PAGE_SIZE;

        }
        ioctl(dev_fd, 0x12345676, kernel_address);
        munmap(kernel_address, PAGE_SIZE);
        break;
```

等待device 時會設定 PAGESHIFT的功能，所以先開啟一個傳輸的裝置位置，
也就是 kernel address . Offset 會是page_size 的倍數以避免資料放入memory map 的時候出現衝突，然後在設定以 page_size為大小的 frame 將一份資料分成一個一個page 在傳輸，當最後一段資料小於page_size，則設定剩餘的size為傳輸資料位址的長度。透過 memcpy

做到 copy from file address 到 kernel address， 然後回傳端口完成。等到所有資料傳輸到 master device 後， 再刪除 kernel address memory map 中的位置。

For slave.c to receive data using mmap, we use this method below:

```
case 'm'://mmap
        kernel_address = mmap(NULL, PAGE_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, dev_fd, 0);
        do
        {
                ret = ioctl(dev_fd, 0x12345678, sizeof(buf));
                //posix_fallocate(file_fd, file_size, ret);

                ftruncate(file_fd, file_size + ret);
                //ftruncate(file_fd, set_size + size_mmap);
                //kernel_address = mmap(NULL, ret, PROT_READ, MAP_SHARED, dev_fd, offset);
                offset = (set_size / PAGE_SIZE) * PAGE_SIZE;
                size_t offsize = set_size % PAGE_SIZE;
                file_address = mmap(NULL, ret, PROT_WRITE, MAP_SHARED, file_fd, offset);

                page_dicrs[page_num] = file_address;
                page_num += 1;

                memcpy(file_address + offsize, kernel_address, ret);
                //munmap(kernel_address, ret);
                ioctl(dev_fd, 0x12345676, file_address);
                munmap(file_address, ret);
                set_size += ret;
                file_size += ret;
                if (i == (file_num - 1)){
                        continue;
                }
                else if (set_size > (1 * PAGE_SIZE)){
                        ret = 0;
                }
        }while(ret > 0);
        ftruncate(file_fd, file_size);
        munmap(kernel_address, PAGE_SIZE);
        break;
```

為了從 slave device 獲得資料，要開啟控制端口，以 BUF_SIZE為單位逐步獲取回傳資料。接下來，以BUF_SIZE的長度修改寫出file size的大小，一樣設定好 file address 的位址，且 copy kernel address 的資料到 file address。當回傳單位資料小於等於0時，已結束獲得所有的資料，然後output 接收到的資料檔案。

Added mmap function can be referred to master_device.c, in slave_device we just change master_open & master_close to slave_open & slave_close

Based on “3/9/10 Reference”

```
static int my_mmap(struct file *filp, struct vm_area_struct *vma);
void mmap_open(struct vm_area_struct *vma) { /*nothing*/ }
void mmap_close(struct vm_area_struct *vma) { /*nothing*/ }
---
static int my_mmap(struct file *filp, struct vm_area_struct *vma)
{
    vma->vm_pgoff = (virt_to_phys(filp->private_data)) >> PAGE_SHIFT;
    if (remap_pfn_range(vma, vma->vm_start, vma->vm_pgoff, vma->vm_end -
vma->vm_start, vma->vm_page_prot))
        return -EIO;
    vma->vm_flags |= VM_RESERVED;
    vma->vm_private_data = filp->private_data;
    vma->vm_ops = &mmap_vm_ops;
    mmap_open(vma);
    return 0;
}
---
int master_close(struct inode *inode, struct file *filp)
{
    kfree(filp->private_data);
    return 0;
}

int master_open(struct inode *inode, struct file *filp)
{
    filp->private_data = kmalloc(PAGE_SIZE, GFP_KERNEL);
    return 0;
}
```

5. RESULT & ANALYSIS

Show the results and explain the performance difference between File I/O and Memory-mapped I/O.

- F and M refer to File I/O and Memory-mapped I/O, respectively.
- PAGE SIZE is 4096 bytes or 4 KB. BUF_SIZE is 512 bytes.

1. We test on sample input files, using 10 small files and one large file. We conducted each experiment 50 times to get a more accurate result. We also measured the standard deviation, as a measure of how far each observed value is from the average.
2. We use Matlab to calculate the average transmission time and use the function "rmoutliers" to detects outliers. By default, an outlier is a value that is more than three scaled median absolute deviations.

- **SAMPLE INPUT 1** (10 small files, 3018 bytes)

MASTER TYPE	**SLAVE TYPE**	**AVERAGE TRANSMISSION TIME** (ms)	**STANDARD DEVIATION**
F	**F**	0.08221538	0.04710715
F	**M**	0.07348824	0.02502600
M	**F**	0.11711250	0.09388420
M	**M**	0.15561400	0.13903968

- **SAMPLE INPUT 2** (1 large file, 1502860 bytes)

MASTER TYPE	**SLAVE TYPE**	**AVERAGE TRANSMISSION TIME** (ms)	**STANDARD DEVIATION**
F	**F**	27.47371622	3.15245884
F	**M**	31.63562333	6.66992326
M	**F**	28.50221714	4.05015185
M	**M**	24.98307059	6.86932763

Compare the performance of File I/O & Memory-mapped I/O

Based on the result we get, after testing it with 10 small files, we came to the conclusion that using mmap will not make the transmission speed faster than using fcntl. In fact, it can even be slower. For smaller data transfer, fcntl may be more efficient due to mmap's large overhead for a single operation in kernel access. In addition, TLB miss and page-fault are also resource-expensive and slow, and that is how the mapping gets populated. Memory-mapped I/O will take time to copy the whole page at once.

For large enough files, memory-mapped I/O is more efficient. Theoretically, as the file grows larger, the difference in performance between memory-mapped I/O and file I/O would also grow bigger too (memory-mapped I/O being the faster one). This is due to mmap's capability of reducing the number of copies of data made. Mmap I/O doesn't require a copy of the file data from kernel to user-space, reducing the number of disk I/O, thus reducing the total transmission time. In addition, memory mapping can simplify the operations by letting the code treat the entire file as accessible.

6. References

1. Posix_fallocate(3) function
2. The Linux Kernel - Memory Mapping
3. http://notailbear.blogspot.com/2008/09/implement-mmap-in-driver.html
4. https://github.com/wangyenjen/OS-Project-2
5. https://github.com/yccyenchicheng/os_project2_sp18
6. https://github.com/b05902046/OS-Project-2
7. https://github.com/qazwsxedcrfvtg14/OS-Proj2
8. https://www.kernel.org/doc/gorman/html/understand/understand006.html
9. Linux Device Driver Development Cookbook page p316~317
10. Linux Device Drivers: Where the Kernel Meets the Hardware p432