2020 OS-Project 2

資工二 趙晉杰 吳庭維 廖榮運 郭瑋喆 陳光裕 徐振棠

1. About Program Design

1-1. Device

根據 Linux Device Drivers, 2nd Edition 中的資訊，我們新增了關於 mmap 以及 VMA Operations 的宣告。

相同的修改處：

```
void mmap_open(struct vm_area_struct *vma) {}
void mmap_close(struct vm_area_struct *vma) {}
struct vm_operations_struct mmap_vm_ops = {
    .open = mmap_open,
    .close = mmap_close
};
```

(這些 operations 是在 kernel 處理完該做的事才會呼叫，然而這次 project 中沒有要求額外的功能，所以留空)

```
static int my_mmap(struct file *filp, struct vm_area_struct *vma)
{
    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;
}
```

只有 master_device.c 中的修改處：

```
case master_IOCTL_MMAP:
    ret = ksend(sockfd_cli, file->private_data, ioctl_param, 0
    break;
```

只有 slave_device.c 中的修改處：

```
case slave_IOCTL_MMAP: // similar to master_device
    ret = krecv(sockfd_cli, file->private_data, PAGE_SIZE, 0);
    break;
```

1-2. Master

利用兩層的 while loop，以 offset 作為傳送的起點。首先利用 mmap 將 input file 映射至指標 src 所指的記憶體位置，一次請求的記憶體空間為 PAGE_SIZE。接下來同樣用 mmap 將虛擬硬體 master device 的記憶體映射至 dst 所指的位置。
接下來不斷用 BUF_SIZE 把 src 的 data 轉移至 dst 並同時更新 offset，一旦 offset 等於 PAGE_SIZE(必定為 BUF_SIZE 的整數倍)或超過 file size，就進行一次 ioctl 並 release 掉所請求的 mapped space，如果檔案還沒傳輸完畢就再次進入迴圈。

```
case 'm': //mmap
    while (offset < file_size) {
        if((src = mmap(NULL, PAGE_SIZE, PROT_READ, MAP_SHARED, file_fd, offset)) == (void *) -1) {
            perror("mapping input file");
            return 1;
        }
        if((dst = mmap(NULL, PAGE_SIZE, PROT_WRITE, MAP_SHARED, dev_fd, offset)) == (void *) -1) {
            perror("mapping output device");
            return 1;
        }
        do {
            int len = (offset + BUF_SIZE > file_size ? file_size % BUF_SIZE : BUF_SIZE);
            memcpy(dst, src, len);
            offset += len;
            ioctl(dev_fd, 0x12345678, len);
        } while (offset < file_size && offset % PAGE_SIZE != 0);
        ioctl(dev_fd, 0x12345676, (unsigned long)src);
        munmap(src, PAGE_SIZE);
    }
    break;
```

1-3. Slave

```
case 'm': //mmap
    while ((ret = read(dev_fd, buf, sizeof(buf))) > 0)
    {
        if (file_size % mmap_size == 0) {
            if (file_size) {
                ioctl(dev_fd, 0x12345676, (unsigned long)dst);
                munmap(dst, mmap_size);
             }
            ftruncate(file_fd, file_size+mmap_size);
            if((dst = mmap(NULL, mmap_size, PROT_READ | PROT_WRITE
             , MAP_SHARED, file_fd, file_size)) == (void *) -1) {
                perror("mapping output file");
               return 1;
            }
        }
        memcpy(&dst[file_size%mmap_size], buf, ret);
        file_size += ret;
    };
    ftruncate(file_fd, file_size);
    ioctl(dev_fd, 0x12345676, (unsigned long)dst);
    munmap(dst, mmap_size);
    break;
```

同樣利用兩層 while loop，用 read 讀取 slave_device 的資料直到 mmap_size(buf 的整數倍)，再轉移至 output file 映射的記憶體位置 dst，因為實在不知道 slave device 端到 kernel 的 mmap 怎麼實作，所以做了半套(kernel→user space)。

時間分析：

經過測試之後發現 fcntl 在第一次跟第二次的傳輸時間有明顯落差，mmap 卻只有最後一筆大資料出現明顯的差別，測試結果如下

	第一次 /第二次		
Mmap:			
10	0.051900ms	0.040500ms	1bytes
101	0.040000ms	0.042600ms	12bytes
1959	0.040200ms	0.062900ms	244bytes
22068	0.114400ms	0.101700ms	2758bytes
221737	0.338600ms	0.350400ms	27717bytes
22205509	1989.860300ms/17.462500ms		2775688bytes
Fcntl:			
10	0.082000ms	0.036600ms	1bytes
101	921.658800ms	0.034600ms	12bytes
1959	0.147000ms	0.089000ms	244bytes
22068	31.131000ms	0.051200ms	2758bytes
221737	0.190200ms	0.761600ms	27717bytes
22205509	981.849300ms	968.269200ms	2775688bytes

我認為 fcntl 會在第二次輸出有明顯落差是因為有 cache 的幫忙，才會有明顯的時間差，而 mmap 沒有，所以時間基本上都差不多。

不過有一個有趣的現象，就是兩者都在最後的大筆資料出現明顯的不同，mmap 在最後出現明顯落差，fcntl 卻反常的時間差不多，經過思考我仍然想不到甚麼原因會造成此種情況。

分工表

B07902120 資工二 趙晉杰：user program & 時間分析
B07902102 資工二 吳庭維：device & report 程式架構
B07902094 資工二 廖榮運：device
B07902056 資工二 郭瑋喆：report 美編
B07902072 資工二 陳光裕：report 美編
B07902032 資工二 徐振棠：report 美編