# Operating System: Project 2

b06902034 黃柏諭
b06902045 張道然
b07902047 羅啓帆
b07902133 黃于軒
b07902139 鄭豫澤
b07902141 林庭風

June 24, 2020

## 1 Design

### 1.1 Device

我們處理`fcntl`的部份基本上是延襲助教的程式碼。另一方面，我們處理`mmap`的方式則是產生出一塊shared memory，再透過`ioctl`來傳遞「送出目前的buffer」或是「populate the buffer with new data」的請求。如此設計的好處是能夠避免多次`mmap`帶來的overhead。

另外，在kernel中allocate該塊memory的方式是使用`alloc_pages`而非`kmalloc`。這是因為`mmap`基本上也是以page作為單位，如此實作希望能夠降低bookkeeping的overhead。

我們存儲allocate的page的方式是直接使用全域變數。這麼做而非使用`private_data`的原因是助教提供的程式碼已使用了非常多的全域變數，導致device已經無法被多個process使用anyway。

關於master和slave device的細項設計會在下面的段落做說明。

#### 1.1.1 Master device

我們在`master_device.c`中增加了以下的程式碼。在`master_mmap`函式中，使用`VM_RESERVED`的flag避免記憶體被swap out，然後透過`page_to_pfn`取得frame number並使用`remap_pfn_range`將kernel space的記憶體map到user space的記憶體中。

另外，這裡判斷了使用者mmap的length不可超過預先allocate的buffer大小，以避免安全性問題的發生。

注意在這裡我們會一次將所有需要map的page map進來，因此無須覆寫`vma->vm_ops`。

此外，若master device收到來自master傳送`master_IOCTL_MMAP`的訊息，就用`ksend`將buffer的資料傳送給slave device。

```
static struct page *buffer;

static struct file_operations master_fops = {
    .owner = THIS_MODULE,
    .unlocked_ioctl = slave_ioctl,
    .open = slave_open,
    .read = receive_msg,
    .release = slave_close,
    .mmap = master_mmap
};

int master_mmap(struct file *filp, struct vm_area_struct *vma) {
    if (vma->vm_end - vma->vm_start > MMAP_BUF_SIZE) {
        pr_err("mmap requested size too large, aborting...");
    }
    vma->vm_flags |= VM_RESERVED;
    int pfn = page_to_pfn(buffer);
    int ret = remap_pfn_range(vma, vma->vm_start, pfn, vma->vm_end - vma->vm_start, vma->
    vm_page_prot);
    if (ret < 0) {
```

```
        return -EIO;
    }
    printk("%lX\n", buffer->flags);
    return 0;
}
```

```
// static int __init master_init(void)
buffer = alloc_pages(GFP_KERNEL, MMAP_BUF_PAGES_LOG);
```

```
// static void __exit master_exit(void)
__free_pages(buffer, MMAP_BUF_PAGES_LOG);
```

```
// static long master_ioctl(struct file *file, unsigned int ioctl_num, unsigned long
    ioctl_param)
case master_IOCTL_MMAP:
    data_size = ioctl_param;
    ret = ksend(sockfd_cli, page_to_virt(buffer), data_size, 0);
    break;
```

### 1.1.2 Slave device

slave_mmap的程式碼部份和master_mmap非常相似。同樣地，若slave device收到來自slave傳送slave_IOCTL_MMAP的訊息，就用krecv接收從master device傳遞過來的資料。

```
case slave_IOCTL_MMAP:
    ret = krecv(sockfd_cli, page_to_virt(buffer), MMAP_BUF_SIZE, 0);
    break;
```

## 1.2 User program

### 1.2.1 master

在master.c中，一開始先把master device map進記憶體中，在迴圈裡，每一次都把長度為MMAP_BUF_SIZE的file map進記憶體中，如果剩餘長度不夠長就把剩下的資料都map進來，接著就把資料透過memcpy複製到device裡並更新offset，最後再進行unmap，直到整個檔案都複製完成。

```
char *dev_mem = mmap(NULL, MMAP_BUF_SIZE, PROT_WRITE, MAP_SHARED, dev_fd, 0);
while (offset < file_size) {
    int64_t remain = file_size - offset;
    int64_t len = remain < MMAP_BUF_SIZE ? remain : MMAP_BUF_SIZE;
    char *file_mem =
        mmap(NULL, len, PROT_READ, MAP_SHARED, file_fd, offset);
    offset += len;
    memcpy(dev_mem, file_mem, len);
    assert(ioctl(dev_fd, MASTER_IOCTL_MMAP, len) >= 0 &&
           "mmap sending failed");
    munmap(file_mem, len);
}
```

### 1.2.2 slave

在slave.c中，先把slave device map進記憶體中，在迴圈中每次利用ioctl讀取device的檔案，如果已經讀完了就離開迴圈。接著就把輸出檔增加長度並且mapping進memory內，然後把dev_mem中的資料複製進file_mem中，再將file的記憶體歸還並更新file_size。

```
int64_t ret = 0;
char *dev_mem =
    mmap(NULL, MMAP_BUF_SIZE, PROT_READ, MAP_SHARED, dev_fd, 0);
assert(dev_mem != (void*)-1);
```

```
for (;;) {
    assert((ret = ioctl(dev_fd, SLAVE_IOCTL_MMAP)) >= 0 &&
           "mmap receiving failed");
    if (!ret) break;
    assert(ftruncate(file_fd, file_size + ret) == 0);
    int64_t extra = file_size % PAGE_SIZE;
    char *file_mem =
        mmap(NULL, extra + ret, PROT_WRITE, MAP_SHARED, file_fd, file_size - extra);
    assert(file_mem != (void*)-1);
    memcpy(file_mem + extra, dev_mem, ret);
    munmap(file_mem, extra + ret);
    file_size += ret;
}
```

### 1.3 Arguments

我們作為測試輸入的argument設計如下

```
./master [-fma] [files]...
./slave [-fma] ip_address [files]...
```

其中`-f`表`fcntl`，`-m`表`mmap`，而`-a`表async+`mmap`。

如此的設計相較於原本的版本比較接近Unix的convention。除此之外，這樣也可以避免計算一共輸入了多少個filename。例如，當我們一次測試一整個資料夾的時候就不用再去計算資料夾內有多少個檔案。

## 2 Results

在我們的實驗中，我們使用了下列不同大小的由`/dev/urandom`產生的檔案。由於前述我們`mmap`的實作能夠降低`mmap`被呼叫的頻率，我們相信我們不管在何種檔案大小下均是相對有優勢的。

我們實驗出來的數據如下(四捨五入至小數下第三位)，前面五個資料夾皆包含8個檔案。

| mean time(ms) / method \ file directory | 1B | 16B | 256B | 4096B | 65536B | 1M | 16M | 4096M |
|---|---|---|---|---|---|---|---|---|
| fcntl | 6.751 | 3.668 | 9.158 | 8.771 | 13.324 | 12.330 | 109.357 | 41692.074 |
| mmap | 1.892 | 2.156 | 1.873 | 1.995 | 5.072 | 5.998 | 63.262 | 35934.990 |

| std of time(ms) / method \ file directory | 1B | 16B | 256B | 4096B | 65536B | 1M | 16M | 4096M |
|---|---|---|---|---|---|---|---|---|
| fcntl | 0.232 | 0.482 | 0.545 | 0.571 | 0.624 | 0.706 | 0.843 | 2.226 |
| mmap | 0.012 | 0.037 | 0.054 | 0.063 | 0.072 | 0.082 | 0.097 | 0.796 |

Flags in the page table descriptor of the `mmap`ed page:


## 3 Discussion

我們在同一台機器上進行測試以避免網路造成的影響。可以發現，一般來說使用mmap進行檔案傳送速度會比fcntl還要快，這是因為mmap在資料讀寫上是直接在memory上進行複製，相比使用fcntl少了一次buffer複製的動作，並且作業系統負責決定什麼時候進行disk flush，相比read/write操作減少了disk I/O的發生次數，因此讓mmap比fcntl還要有效率。

但是在檔案很小的情況下，mmap和fcntl的時間相去不遠，甚至有一定概率時候mmap會慢一些，由於小檔案傳輸時間本來就很短，而且mmap需要建立page table和TLB，如果遇到page fault或TLB miss就會造成更多的 memory access，所以有可能mmap在這種時候就會比較慢。

# 4 Bonus

作為bonus，我們實作了asynchronous的介面。

在interface上，我們提供了兩個新的`ioctl`：`MMAP_ASYNC_COMMIT`和`MMAP_ASYNC_CHECK`，作為`MMAP`的asynchronous版本。

使用上，使用者可以呼叫`MMAP_ASYNC_COMMIT`，通知device處理當前buffer，並使用`MMAP_ASYNC_CHECK`確認commit是否完成。

實作上，我們內部使用了`async_schedule`來launch send或receive的task，並在task完成後修改一個volatile的變數，以讓`MMAP_ASYNC_CHECK`得知工作的完成。

在user space下，由於我們工具比較單純，無法體現出asynchronous I/O的優勢（於等待時進行其他的工作），我們只能在送出commit request之後進行等待，每1$\mu$s進行一次檢查，直至工作完成。

## 4.1 Results

我們使用async+mmap來傳送資料，在傳送16M的資料夾花了343ms，比synchronous+mmap的傳送方式還要慢。這主要是因為前述的原因。另一方面，若application的性質能體現出asynchronous的優勢，相信會得到更好的效能。

# 5 Work Loads

- b06902034 黃柏諭：整理bonus實作方式並進行和sync結果比對。
- b06902045 張道然：生成大檔案，結果比對與報告撰寫。
- b07902047 羅啓帆：實驗測試與結果分析，報告撰寫。
- b07902134 黃于軒：device, user program, bonus程式撰寫。
- b07902139 鄭豫澤：整理device部份實作程式碼與結果分析。
- b07902141 林庭風：整理user program實作程式碼，報告撰寫。

# 6 Reference

- `https://linux-kernel-labs.github.io/refs/heads/master/labs/memory_mapping.html`
- 助教們提供的sample code.