OS Final Report

Members

資工二 B07902006 林則仰

資工二 B07902013 陳建豪

資工二 B07902016 林義閔

資工二 B07902021 簡捷

資工二 B07902060 趙雋同

資工二 B07902114 陳柏衡

Programming Design

Device

我們主要在 slave device 以及 master device 中新增 mmap 的部分。

- slave device 的 mmap: 實作 slave_mmap，code 如下：

```
int slave_mmap(struct file *filp, struct vm_area_struct *vma){
    unsigned long pfn_start = virt_to_phys(filp->private_data) >> PAGE_SHIFT;
    unsigned long size = vma->vm_end - vma->vm_start;
    if(remap_pfn_range(vma, vma->vm_start, pfn_start, size, vma->vm_page_prot))
{
        printk("remap_pfn_range failed at [0x%lx  0x%lx]\n", vma->vm_start,
vma->vm_end);
        return -EAGAIN;
    }
    vma->vm_ops = &simple_remap_vm_ops;
    vma->vm_flags |= VM_RESERVED;
    vma->vm_private_data = filp->private_data;
    vma_open(vma);
    return 0;
}
```

在這個函式中，先將 virtual address (logical address)利用 virt_to_phys 轉換成physical address，接著算出所需要的size後，利用剛剛得到的 physical address 向 kernal 要相對應的 page。成功之後，接著定義這塊記憶體上的 operation、flag、以及這塊記憶體對應到的檔案內容。

simple_remap_vm_ops 只有簡單的 open 跟 close（定義了記憶體的 operation），程式碼如下：

```
void vma_open(struct vm_area_struct *vma){ return ; }
void vma_close(struct vm_area_struct *vma) { return; }
static const struct vm_operations_struct simple_remap_vm_ops
{
    .open = vma_open,
    .close = vma_close,
}
```

- master device 的 mmap: 實作 master_mmap, 程式碼如下：

```
static int master_mmap(struct file *file, struct vm_area_struct *vma)
{
    unsigned long pfn_start = virt_to_phys(file->private_data) >> PAGE_SHIFT;
    unsigned long size = vma->vm_end - vma->vm_start;
    if(remap_pfn_range(vma, vma->vm_start, pfn_start, size, vma->vm_page_prot))
{
        printk("remap_pfn_range failed at [0x%lx  0x%lx]\n", vma->vm_start,
vma->vm_end);
        return -EAGAIN;
    }
    vma->vm_ops = &simple_remap_vm_ops;
    vma->vm_private_data = file->private_data;
    vma->vm_flags |= VM_RESERVED;
    vma_open(vma);
    return 0;
}
```

這一段 mmap 的實作和 slave device 的 mmap 基本上相同，simple_remap_vm_ops 也和 slave device 用到的一樣。

Master Program

在 master.c 中，我們新增了處理mmap的部分。在這部分的實作中，我們定義offset為目前傳送的byte數，再定義要傳送的資料長度，若目前 offset 加一個PAGE_SIZE後仍沒有超出file的大小，就定義要傳送的資料的長度send_len=PAGE_SIZE，若剩餘的長度不足定義為file_size - offset，也就是只 map 剩下的長度。接著，master 將 target file 以及 master device 都分別利用 mmap 來對應到一塊記憶體（input_mem & output_mem），接著利用 memcpy 將一段長度為send_len的資料將target file的內容寫入master device。寫入之後，利用ioctl函式與master device 溝通，使 master device 利用 ksocket 中的ksend將資料傳送給slave device。最後在寫入完成後，將input_mem以及output_mem兩塊記憶體利用munmap函式歸還memory。

mmap 部分的程式碼如下：

```
size_t offset = 0;
int send_len;
char *input_mem;
char *output_mem;
while(offset < file_size){
```

```
    send_len = ((file_size - offset) > PAGE_SIZE)? PAGE_SIZE : (file_size -
offset);
    input_mem = mmap(NULL, send_len, PROT_READ, MAP_SHARED, file_fd, offset);
    output_mem = mmap(NULL, send_len, PROT_WRITE, MAP_SHARED, dev_fd, offset);
    memcpy(output_mem, input_mem, send_len);
    send_len = ioctl(dev_fd, 0x12345678, send_len);
    offset += send_len;
    munmap(input_mem, send_len);
    munmap(output_mem, send_len);
}
```

Slave Program

在 slave.c 中，我們一樣新增了 mmap 的實作，一開始 slave 先利用ioctl來通知 slave device 去接收 ksocket 傳來的資料，並定義 recv_len 為資料的長度。接著，由於 mmap 無法 map 到沒有資料的記憶體，因此必須先利用 ftruncate 來將檔案的大小從原先的 file_size 加大至 file_size+recv_len 。如此一來，就可以仿效master.c的方法，先將slave device 以及 received file 都 map 到一塊記憶體（input_mem & output_mem），接著再利用memcpy將一段長度為 recv_len 的資料從 slave device 寫入 received file，從而實現從 slave device寫入 received file的功能。最後在寫入完成後，將input_mem以及output_mem兩塊記憶體利用munmap函式歸還memory。

mmap 部分的程式碼如下：

```
int recv_len;
char *input_mem;
char *output_mem;
while((recv_len = ioctl(dev_fd, 0x12345678, file_size)) != 0){
    if(recv_len < 0){
        perror("receive message error\n");
        return 1;
    }
    ftruncate(file_fd, file_size + recv_len);
    input_mem = mmap(NULL, PAGE_SIZE, PROT_READ, MAP_SHARED, dev_fd,
file_size);
    output_mem = mmap(NULL, PAGE_SIZE, PROT_WRITE, MAP_SHARED, file_fd,
file_size);
    memcpy(output_mem, input_mem, recv_len);
    munmap(input_mem, PAGE_SIZE);
    munmap(output_mem, PAGE_SIZE);
    file_size += recv_len;
}
```

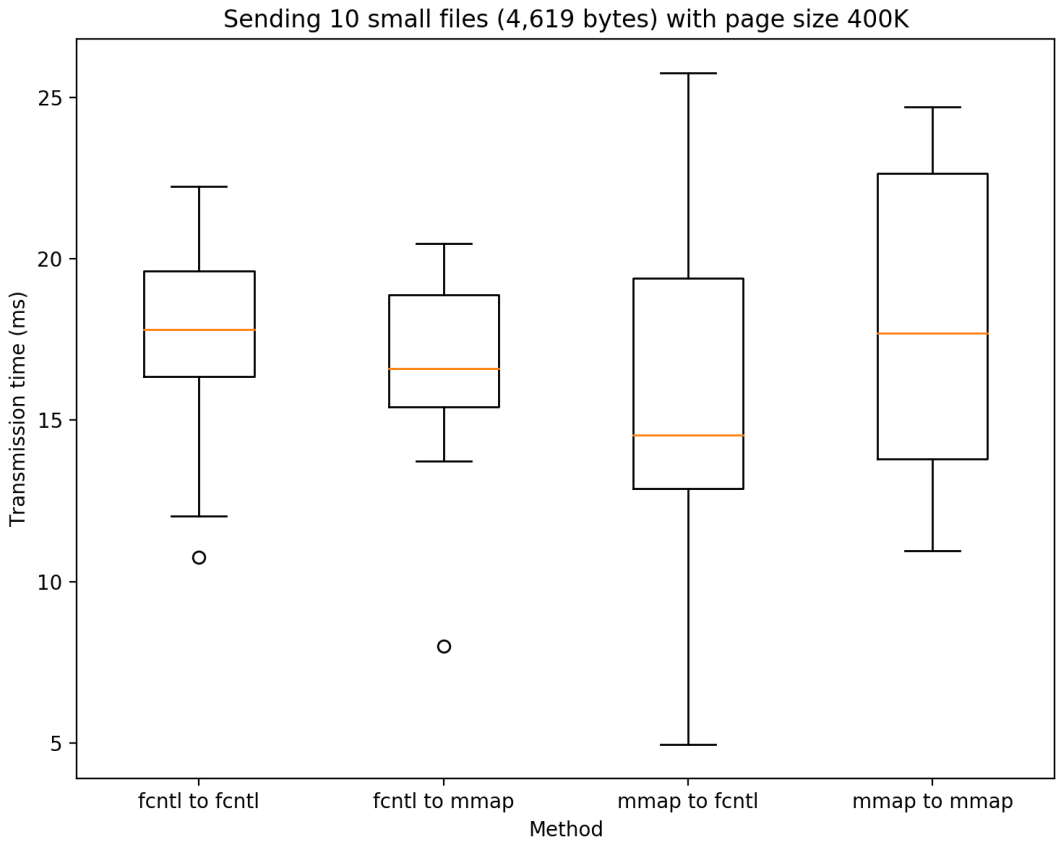
Problem and solution

一開始在實作 mmap 時，我們觀察到一種 race condition：如果 master device 用 fcntl傳資料，slave device 用 mmap 收資料，可能會發生 master device 還沒傳輸足夠的資料（小於 4K）時，slave device 就去接收資料，接下來的 mmap 就會發生 segmentation fault，因為 mmap 的最後一個 offset 參數變得不是 page size（4K）的整數倍。為了修復這個問題，我們在 slave device 收資料的部分使用

while 迴圈控制，保證 slave device 每次收到的資料和回傳值都是 page size 的整數倍來解決這個 race condition。

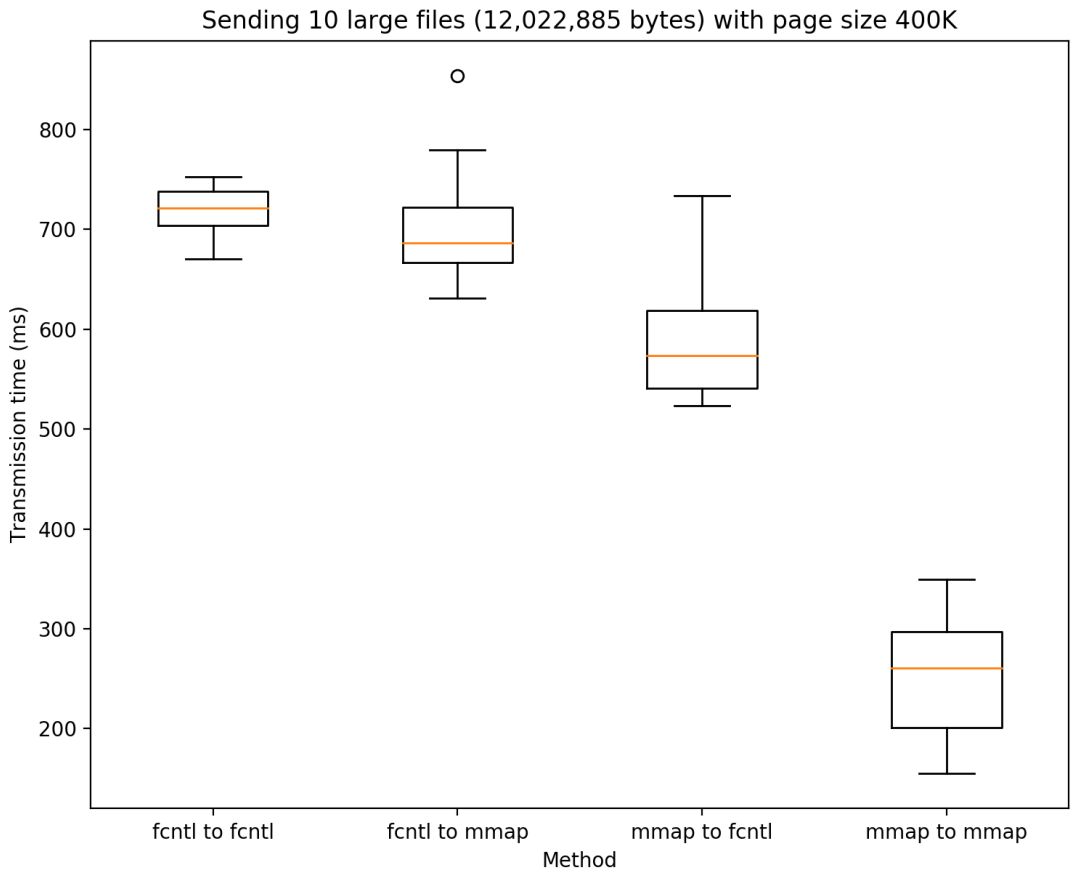
Result and comparison

The result



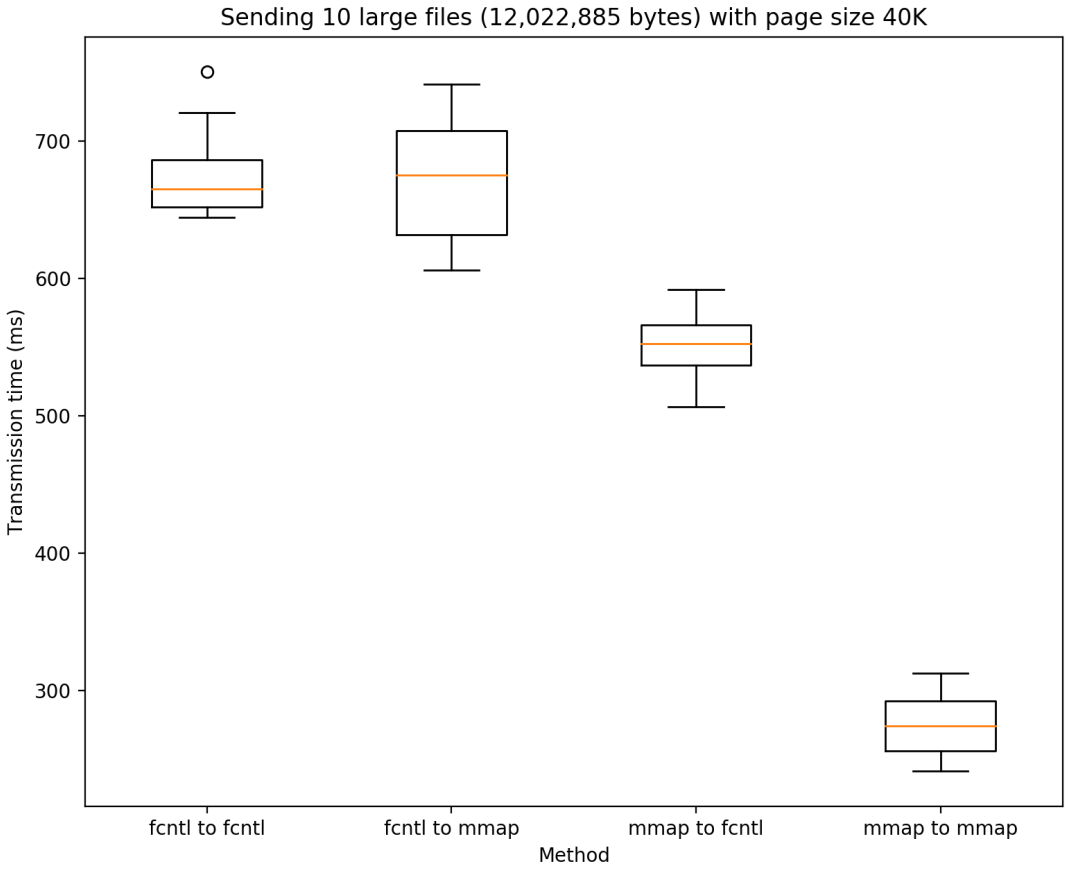
表格 1: 使用四種方法傳送 10 個小檔案之比較（page size 400K）

Page Size	檔案	Slave	Master	平均	標準差
400K	10 小	Fcntl	Fcntl	17.259	3.105
400K	10 小	Fcntl	Mmap	16.531	3.178
400K	10 小	Mmap	Fcntl	15.753	5.484
400K	10 小	Mmap	Mmap	17.870	4.738



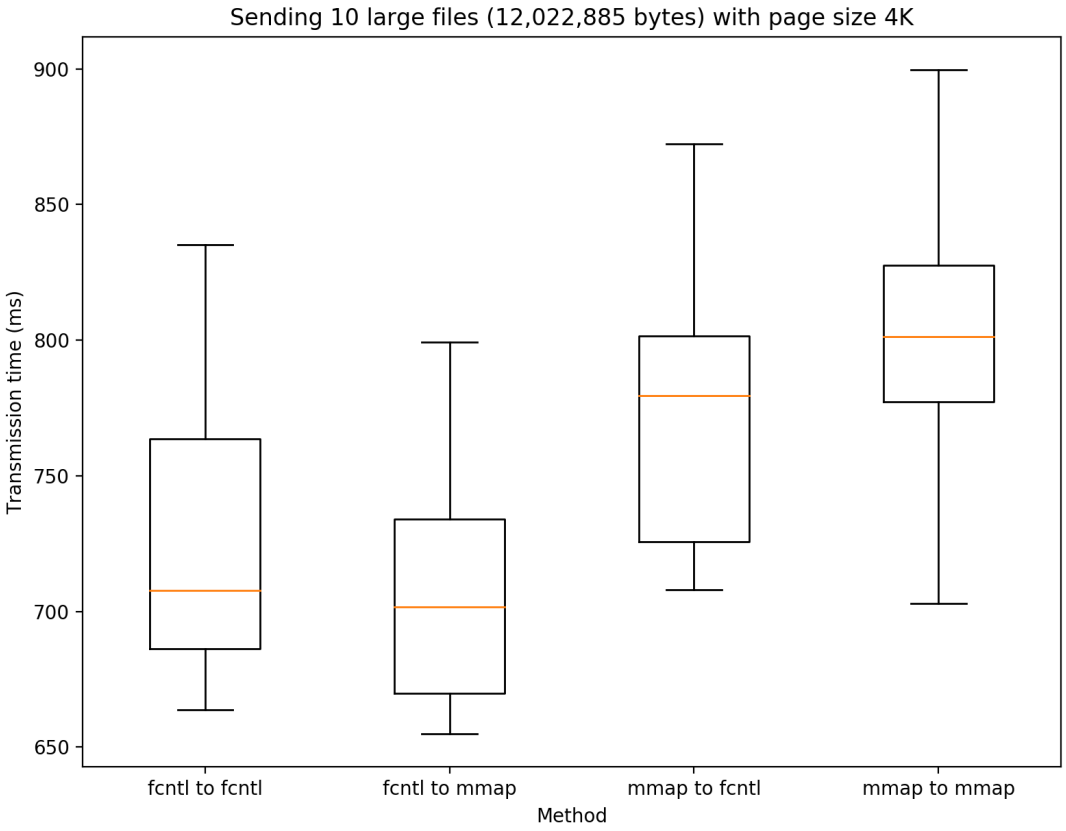
表格 2: 使用四種方法傳送 10 個大檔案之比較（page size 400K）

Page Size	檔案	Slave	Master	平均	標準差
400K	10 大	Fcntl	Fcntl	716.162	25.952
400K	10 大	Fcntl	Mmap	701.895	56.147
400K	10 大	Mmap	Fcntl	586.362	59.731
400K	10 大	Mmap	Mmap	252.450	56.449



表格 3: 使用四種方法傳送 10 個大檔案之比較（page size 40K）

Page Size	檔案	Slave	Master	平均	標準差
40K	10 大	Fcntl	Fcntl	676.057	29.843
40K	10 大	Fcntl	Mmap	672.965	46.146
40K	10 大	Mmap	Fcntl	552.962	24.297
40K	10 大	Mmap	Mmap	274.503	22.265



表格 4: 使用四種方法傳送 10 個大檔案之比較（page size 4K）

Page Size	檔案	Slave	Master	平均	標準差
4K	10 大	Fcntl	Fcntl	726.957	50.514
4K	10 大	Fcntl	Mmap	707.874	45.113
4K	10 大	Mmap	Fcntl	777.208	48.364
4K	10 大	Mmap	Mmap	802.917	52.057

The comparison between file I/O and memory-mapped I/O

在大的 page size（400K 及 40K）之下，執行的快慢為 mmap to mmap > mmap to fcntl > fcntl to mmap > fcntl to fcntl。尤其是 master 以及 slave 都改成用 mmap 來傳送資料的 case 在速度上大大的勝過其他比較組。這個結果符合我們預期的結果，因為 mmap 在讀寫的時候只會由 memcpy 來做 memory 的複製，disk flush的時機由作業系統來決定，大大減少了原本使用 fcntl 時所做的 I/O次數，因此整體的tranmission time 也會大幅降低。在 master 及 slave 兩端實作 mmap 都能增加使資料的讀寫效率增加。

然而，在非常小的 page size（4K）之下，我們得到了不符預期的結果，mmap 及 fcntl 所花的時間相去不遠，甚至 mmap 還比 fcntl 慢了一些。我們認為這可能是因為 4K 太小，以至於 mmap 方法中建立 mmap 太多次，而每次建立 mmap 來對應 file 和記憶體都會有可觀的 overhead，因此相較於較大的 page size 效率會比較不好。

表格 5: 使用不同大小 page size 之比較

Page Size	檔案	Slave	Master	平均	標準差
4K	10 大	Mmap	Mmap	802.917	52.057
40K	10 大	Mmap	Mmap	274.503	22.265
400K	10 大	Mmap	Mmap	252.450	56.449

Bonus

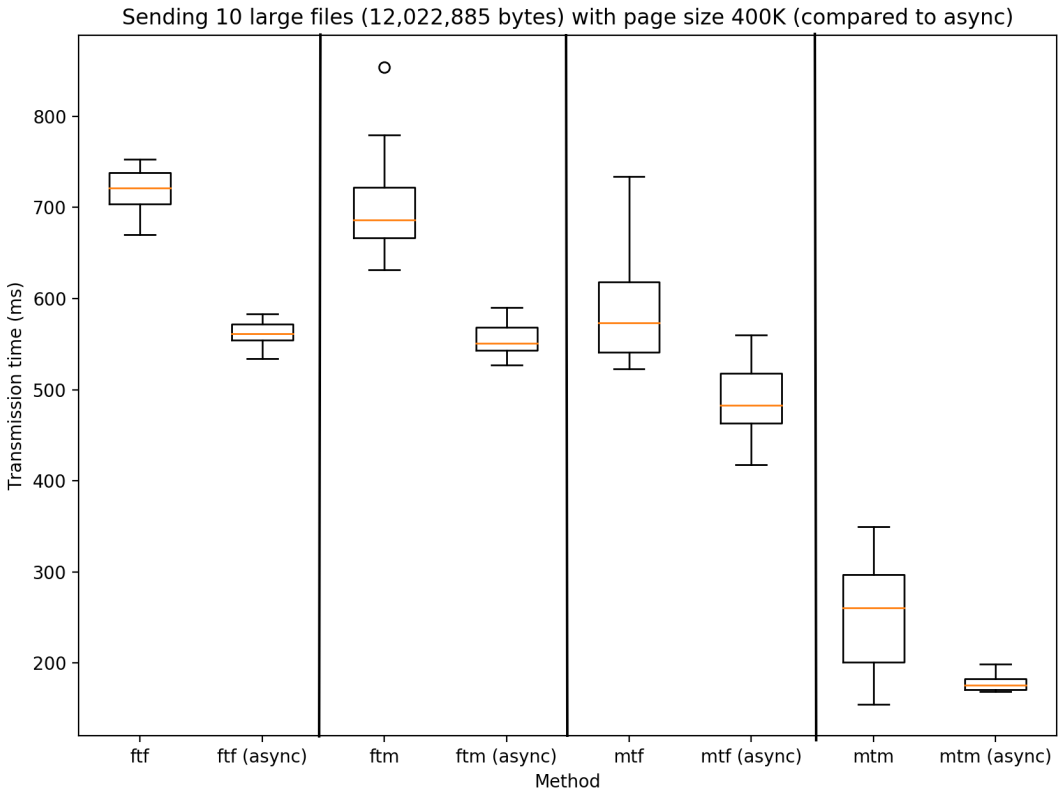
Design

Bonus 的部分，我們將原本的 ksocket 從 sync 版本改至 async 版本， 實作的方法只要將原本 socket 這個 struct 中的 flag 由 `__O_SYNC` 改為 `FASYNC` 即可，也就是在只要在 ksocket 函式加上 `sk->flags |= FASYNC;` ，以及在kaccept函式中加上`new_sk->flags |= FASYNC;`，就成功實現了 asynchronous version 的 kernel socket。

Result and comparison

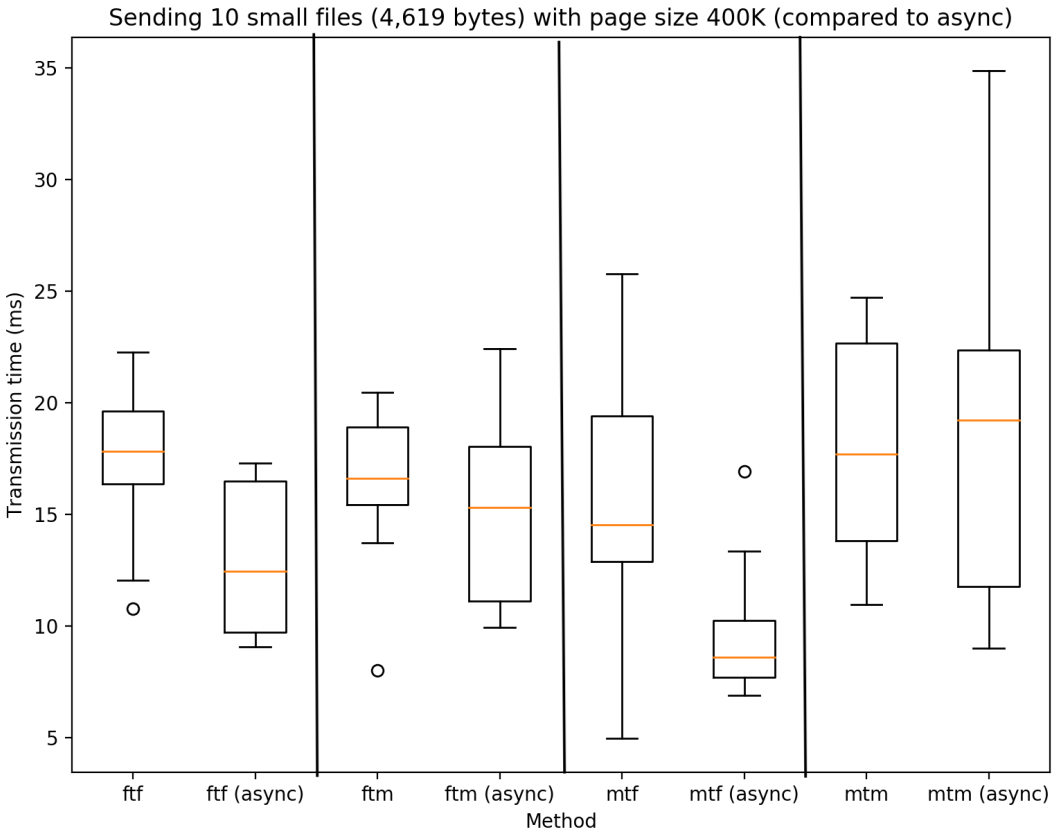
註：以下圖表使用了下列的縮寫

```
ftf: fcntl to fcntl
ftm: fcntl to mmap
mtf: mmap to fcntl
mtm: mmap to mmap
```



表格 6: 使用四種方法傳送 10 個大檔案之比較（async）

Page Size	檔案	Slave	Master	平均	標準差
400K	10 大	Fcntl	Fcntl	561.786	14.077
400K	10 大	Fcntl	Mmap	554.213	18.896
400K	10 大	Mmap	Fcntl	488.087	43.433
400K	10 大	Mmap	Mmap	178.486	9.737



表格 7: 使用四種方法傳送 10 個小檔案之比較（async）

Page Size	檔案	Slave	Master	平均	標準差
400K	10 小	Fcntl	Fcntl	12.991	3.335
400K	10 小	Fcntl	Mmap	15.156	4.183
400K	10 小	Mmap	Fcntl	9.708	3.018
400K	10 小	Mmap	Mmap	18.695	7.513

Discussion

由實驗結果可知 page size 大小為 400K、傳送十個大檔案的 case 之下，在先前我們比較的四種情況中，async version 的 performance 都明顯優於 sync version。這與我們的預測是相符合的，因為 async 版本的 TCP 連線相較於 sync 版本的，程式不會因為等待同步完成的而陷入長時間的等待，而是能有效率的同時接收不只一個 request。

在 page size 為 400K、傳送十個小檔案的 case 之下，可以看到依然與我們先前的預測大致吻合，但是唯獨在 mmap to mmap 這個 case 之下 sync version 的 performance 是略優於 async version 的。會觀察到這樣的現象，我們猜測是因為 async 本身就帶有較多的 overhead，例如會有一些像清空 buffer 較多次而被忽略的 time cost 等等，加上 mmap 本身也有相對於 fcntl 重的 overhead，因此當同時使用 async 以及 mmap to mmap 時，因為上述 overhead 的緣故而使得這樣的組合在小檔案的 case 反而不盡理想。

Conclusion

綜上所述，在理想的情況（page size夠大、傳送檔案夠大）情形下，使用 async version 或是使用 mmap 都能普遍地提升檔案傳輸的效率。但是實際上由於上mmap運作複雜，處理不同檔案有可能會得到非預期中的結果，使用 async socket 亦是如此，因此仍需要針對不同 case 以及較多次的實驗來得到較精準的結果。

Contribution of team members

組員	工作分配	比例
陳建豪	所有的程式部分	20%
林則仰	撰寫 report	16%
林義閔	撰寫 report	16%
簡捷	實驗及整理數據	16%
陳柏衡	實驗及整理數據	16%
趙雋同	實驗及整理數據	16%

Demo

依序demo sample_input_1 with mmap/fcntl, sample_input_2 and large_input(在影片中為 make_data) with mmap/fcntl

Reference

1. 記憶體映射函數mmap 的使用方法@ Welkin小窩
2. 核心記憶體
3. Linux設備驅動remap_pfn_range() 和remap_page_range()
4. Character device drivers -- The Linux Kernel Documentation
5. https://stackoverflow.com/questions/2331869/what-are-async-sockets