2019 OS Project 2
Synchronous Virtual Device Report

資工二 b06902093 王彥仁
資工二 b06902026 吳秉柔
資工二 b06902041 吳采耘
資工二 b06902054 蔡宥杏
醫學三 b05401009 謝德威
電機五 b03901056 孫凡耕

Programming design

Device

我們在 master_device 和 slave_device 中增加 mmap 的部分。

master_device.c 和 slave_device.c 的共同 mmap 部分程式碼

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

static struct file_operations master_fops = {
    .owner = THIS_MODULE,
    .unlocked_ioctl = master_ioctl,
    .open = master_open,
    .write = send_msg,
    .release = master_close,
    .mmap = my_mmap
};


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;
}
```

master_device.c 中 mmap 程式碼

```
ret = ksend(sockfd_cli, file->private_data, ioctl_param, 0);
```

slave_device.c 中 mmap 程式碼

```
ret = krecv(sockfd_cli, file->private_data, PAGE_SIZE, 0);
```

Master

Master 使用 mmap 把檔案"file_fd"mapping 至記憶體中，稱為 src，接著使用將 mmap 後的檔案寫入"dev_fd"。

首先在第一個 while 迴圈中，我們設定 offset 作為下一次傳送的起點。因此在前兩個 if 中，我們利用 mmap 取得一塊 memory，若回傳-1 則代表 mmap 失敗，並顯示錯誤訊息。在接下來的 do-while 迴圈中，我們利用 write 指令將檔案內容從檔案本身的 memory 寫入透過 mmap 獲得的 memory 中，設定 if 條件防止傳送超過檔案大小的內容，並更新 offset。而若 offset 超過檔案大小或是 mmap 的 memory 已滿，便跳出 do-while 迴圈，透過 munmap 指令把 memory 還給系統。而若 offset 的大小未滿檔案大小，則繼續留在第一個迴圈中，繼續下一次傳送。

master.c 中 mmap 程式碼

```
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);
    munmap(src, PAGE_SIZE);
}
```

Slave

Slave 會 read 從"dev_fd"進來的資料，當新資料是達到 mmap_size 之後，會將檔案"file_fd"mmap 一個新的 dst，最後再將 read 近來的 buf 用 memcpy 寫到 dst 裡。

首先在第一個 while 迴圈中，我們設定 ret 作為在 memory 中訊息的大小。在第一個 if 條件中，若目前檔案大小到達 memory 的極限，便 munmap 目前的 memory 並抹去目前檔案大小加上一塊 memory 大小後的檔案內容，然後重新 mmap 一塊新的 memory。之後利用 memcpy 把 memory 中的內容寫入檔案中，並更新檔案大小。最後在 memory 中沒有訊息後，便抹去檔案中超過目前檔案大小的內容，利用 munmap 歸還 memory。

slave.c 中 mmap 程式碼

```
while ((ret = read(dev_fd, buf, sizeof(buf))) > 0)
{
   if (file_size % mmap_size == 0) {
       if (file_size) 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);
munmap(dst, mmap_size);
```

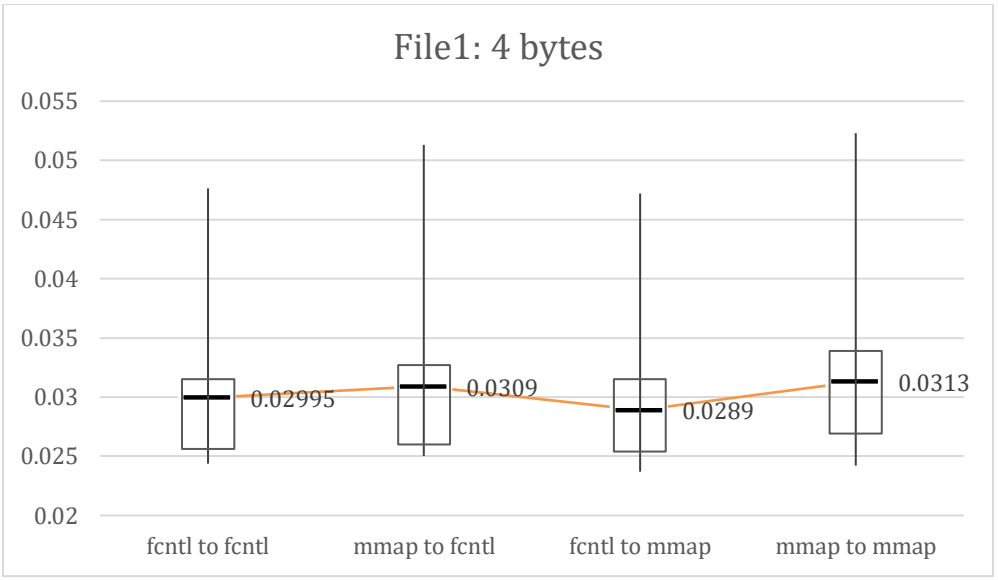
The Result

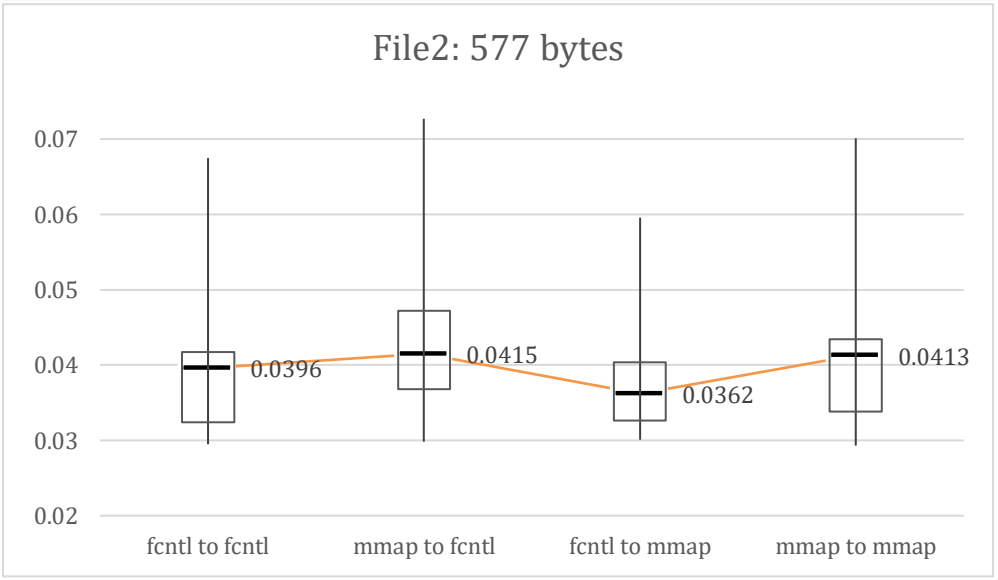
我們將 4 組測試資料分別以 Master、Slave 皆採 fcntl 或 mmap 進行多次測試，扣除極端值後，將有效資料數量、平均值與標準差表列如下：

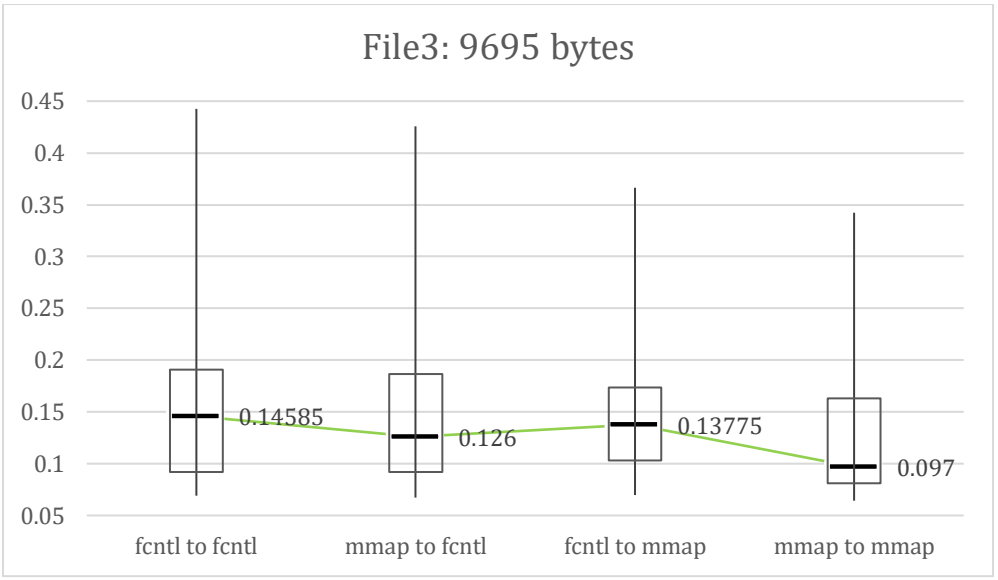
	fcntl to fcntl				mmap to mmap			
	file1_in	file2_in	file3_in	file4_in	file1_in	file2_in	file3_in	file4_in
	4	577	9695	1502860	4	577	9695	1502860
valid data	92	87	98	78	95	94	93	86
average	0.04241	0.05706	0.25220	11.1045	0.04448	0.06059	0.20826	8.29422
std	0.00379	0.00314	0.04190	1.82907	0.00495	0.00542	0.03614	0.69182

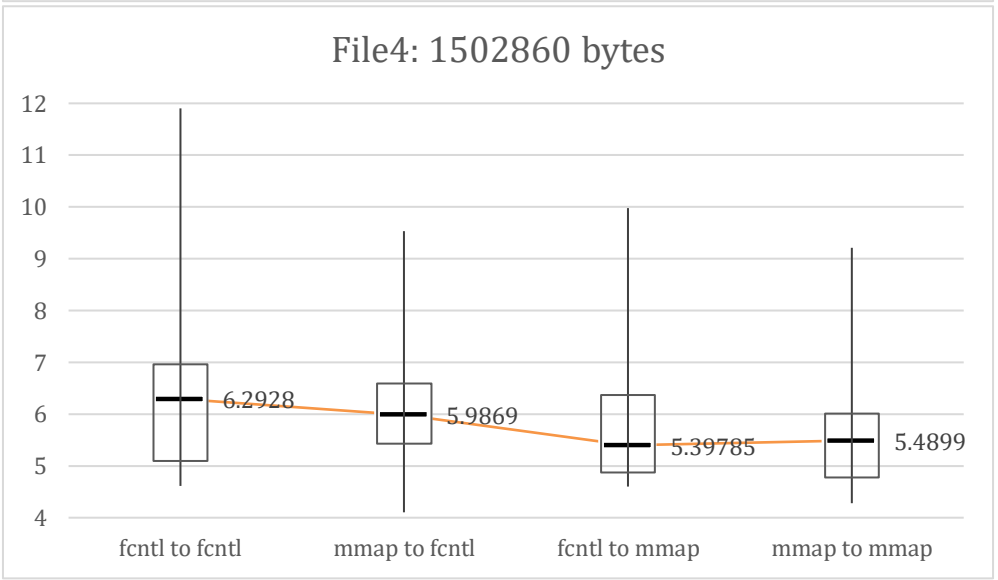
接著將 4 組測試資料分別以 Master、Slave 採用 fcntl 與 mmap 不同搭配進行多次測試，進行統計與繪製盒狀圖如下：

	fcntl to fcntl	mmap to fcntl	fcntl to mmap	mmap to mmap
0.75	0.0315	0.0327	0.0315	0.0339
outlier-high	0.04765	0.0513	0.0472	0.0523
median	0.02995	0.0309	0.0289	0.0313
outlier-low	0.0244	0.025	0.0237	0.0242
0.25	0.0256	0.026	0.0254	0.0269
0.75	0.0417	0.0472	0.0404	0.0434
outlier-high	0.0675	0.0727	0.0596	0.0701
median	0.0396	0.0415	0.0362	0.0413
outlier-low	0.0295	0.0298	0.0301	0.0293
0.25	0.0324	0.0368	0.0326	0.0338
0.75	0.1908	0.1867	0.1738	0.1628
outlier-high	0.44255	0.426	0.36635	0.3421
median	0.14585	0.126	0.13775	0.097
outlier-low	0.0689	0.0674	0.0699	0.0643
0.25	0.0919	0.092	0.103	0.0811
0.75	6.961	6.5885	6.3632	6.0159
outlier-high	11.9031	9.5353	9.97765	9.2081
median	6.2928	5.9869	5.39785	5.4899
outlier-low	4.6115	4.1038	4.5977	4.2846
0.25	5.0909	5.4263	4.871	4.7765


File1: 4 bytes
0.055
0.05
0.045
0.04
0.035
0.03
0.025
0.02
0.02995
0.0309
0.0289
0.0313
fcntl to fcntl
mmap to fcntl
fcntl to mmap
mmap to mmap


File2: 577 bytes
0.07
0.06
0.05
0.04
0.03
0.02
0.0396
0.0415
0.0362
0.0413
fcntl to fcntl
mmap to fcntl
fcntl to mmap
mmap to mmap


File3: 9695 bytes
0.45
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0.14585
0.126
0.13775
0.097
fcntl to fcntl
mmap to fcntl
fcntl to mmap
mmap to mmap


File4: 1502860 bytes
12
11
10
9
8
7
6
5
4
6.2928
5.9869
5.39785
5.4899
fcntl to fcntl
mmap to fcntl
fcntl to mmap
mmap to mmap

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

在非常小的測試資料下(如 file1_in 與 file2_in)，mmap 與 fcntl 所花的時間並沒有顯著的差別，甚至 mmap 會稍微慢一些，這可能是因為 mmap 也有可觀的 overhead，因為需要進行 page table 的建立與 TLB flush 等等，加上如果遇到 TLB miss 甚至 page fault 又會造成更多 memory access 時間的浪費 [1, 2]。

實際上 mmap 是 demand paging 的"lazy" I/O，作業系統將部份的檔案建立 page table 放在記憶體中，如果發現要用的檔案不在記憶體中則需要從 disk 中讀取、進行 replace。但是在較大的資料上(file3_in 與 file4_in)，mmap 就顯著地比 fcntl 來得快。這有可能是因為 mmap 在讀寫時只做了 memory 上的複製，再由作業系統決定 disk flush 的時機，而減少了實際上 disk I/O 的次數與對應的等待時間(根據課本上面的說法，hard disk latency 約為 3ms、而 seek 要花 5ms 再加上 transfer time 0.05ms)。如此 Master 與 Slave 的傳輸皆能變得更有效率。

參見 Linus Torvalds 對於 mmap 效能的看法：

1. https://marc.info/?l=linux-kernel&m=95496636207616
2. http://lkml.iu.edu/hypermail/linux/kernel/0802.0/1496.html

Bonus

Bonus 的部分，我們主要把 sync 的 kernel socket 改成 async 的版本。 kernel socket 的傳輸主要會使用到 socket 這個 structure，其定義如下（定義在 linux/net.h 中）

socket 資料結構

```
struct socket
{
     socket_state  state;
     short type;
     unsigned long flags;
     struct socket_wq __rcu *wq;
     struct file *file;
     struct sock *sk;
     const struct proto_ops *ops;
}
```

其中 flags 的部分可以選擇要 sync 的 socket 或 async 的，其對應的 flags 分別為（定義在 linux/fcntl.h 中）：`__O_SYNC` 和 `FASYNC`。 我們將 ksocket.c 中，每個 socket struct 初始化以後的 flags 加上 `FASYNC` (`sk->flags |= FASYNC;`)即實現了 async 的 socket。

sync 和 async 的結果預測

我們在執行前先猜測 async 的 transmission time 會比 sync 的要少，因為 sync 的要在 buffer 是滿的時候才會一次性清空，而 async socket 則沒有這個要求，因此在執行上 async 應該會比 sync 略快一些。當然，當 buffer 空間不夠大的時候，這結果是不太明顯的。

sync 和 async 的實際結果

因為助教提供的測試資料皆比較小，transmission time 通常不到 100ms，為了讓結果（transmission time 的差異）更加明顯，我們採用了自己生成的較大測試資料來測試。
生成測試資料的程式如下：

生成測試資料

```c
#include <stdio.h>
int main() {
    for (int i = 0; i < 10000000; i++) {
        puts("meow!");
        puts("Cats are so cute!");
    }
    return 0;
}
```

其內容共 2×10^7 行，奇數行皆為字串 `meow!`，而偶數行皆為字串 `Cats are so cute!`

實際結果

	Master	Slave
	fcntl	fcntl
sync	Transmission time: 24996.954500 ms, File size: 275000000	
async	Transmission time: 20998.631500 ms, File size: 275000000	
	fcntl	mmap
sync	Transmission time: 17920.159400 ms, File size: 275000000	
async	Transmission time: 15864.345400 ms, File size: 275000000	
	mmap	fcntl
sync	Transmission time: 18983.638200 ms, File size: 275000000	
async	Transmission time: 19938.115800 ms, File size: 275000000	
	mmap	mmap
sync	Transmission time: **15049.358400** ms, File size: 275000000	
async	Transmission time: 15939.756300 ms, File size: 275000000	

在 master 端是 fcntl 時，async 的 transmission time 普遍會比 sync 的要短；而當 master 端是 mmap 時，async 的 transmission time 會比 sync 的要得長一些。

後者跟我們預期的結果不太一樣，我們認為有可能跟 buffer 的大小有關。當 buffer 大小不夠大時，基本上每次操作都會把 buffer 填滿，因此兩者的行為是差不多的，transmission time 差異不會太大。

此外，async 清空 buffer 的次數可能會比 sync 的略多，我們原本的猜測並未考慮清空 buffer 所造成的時間成本。

Discussion and Conclusion

當我們使用自行生成的測試資料加上 sync 與 async 的比較後得到的結果是當 Master 與 Slave 皆使用 mmap 時，sync 的傳輸方式是最快的。而 Master 使用 fcntl、Slave 使用 mmap 與 Master、Slave 皆使用 mmap 時，使用 async 傳輸的速度相近，且是第二快的組合。

首先我們先看 sync 的 socket 所花的傳輸時間。若 Slave 皆是 fcntl，比較 Master 使用 fcntl 與 mmap 的差異，可以發現 Master 使用 mmap 可以省下約為 24%的時間((24997-18984)/24997)。當 Slave 皆是 mmap 時，Master 使用 mmap 則能省下約為 16%的時間((17920-15049)/17920)。

另一方面，如果固定 Master 使用 fcntl，Slave 使用 mmap 則會比使用 fcntl 快了約為 28%的時間((24997-17920)/24997)！同樣固定 Master 使用 mmap 的話，Slave 使用 mmap 也會比 fcntl 減少約為 21%的時間((18984-15049)/18984)。

綜合起來若 Master 與 Slave 都使用 mmap 的話，可以省下約 40%的時間((24997-15049)/24997)。前者推測可能是因為 mmap 是 demand paging，只會將當前有需要的檔案載入到記憶體中，加上實際進行 disk I/O 的次數可能較少，不如 fcntl 當 buffer 滿了就會進行。Slave 主要的工作是寫入檔案，這裡 memcpy 之後再由作業系統 flush 進 disk 會比一直進行 write 快得多！

如果看 async 的 socket 的話，在 Master 使用 fcntl 時有顯著提昇傳輸效率，但在 Master 使用 mmap 之後就稍微比 sync 慢，這可能是 async 有多出一些 overhead，例如上述 buffer 大小的問題，而 async 的效果在資料是 write 輸出時比較明顯，只要讀資料進來，就可以 non-blocking 地傳送。當資料是 mmap 輸出時，反而因為 async 加上 mmap 的 overhead 造成時間變長。

雖然單一檔案傳輸並不是 mmap 最適合使用時機，而應該是有部份程式、檔案經常性地被使用、更動而需要存在記憶體中，或是多個程式同時需要分享一個檔案時，但 mmap 在這次實驗中仍然在大檔案傳輸勝過 fcntl，也就是一般檔案的 read、write。實際上 mmap 的運作複雜，處理不同檔案與指令可能會遇到非常不一樣的效果，因此要更準確預測執行結果需要多加以實驗與評估。

Contributions of Team Members

資工二 b06902093 王彥仁	Device 與 Bonus 的程式實作與實驗
資工二 b06902026 吳秉柔	Slave 及 Master 的程式實作
資工二 b06902041 吳采耘	整理 Slave 及 Master 的設計與實作方式
資工二 b06902054 蔡宥杏	Slave 及 Master 的程式實作
醫學三 b05401009 謝德威	報告整理、分析與結論撰寫
電機五 b03901056 孫凡耕	Bonus 的程式實作與實驗