Operating System Project 2 Report

設計

`master.c / slave.c`: 我們參考了 這裡 和 這裡 ，在 master 和 slave 的 device driver 裡新增了 mmap 的 code。如此一來，我們可以利用 mmap 將 kernel 的記憶體映設至 user program 的 memory。之後，再將我們從 disk load 進來的檔案搬到 memory 對到 kernel 裡的那塊，達到資料的移動。 另外，我們在其中遇到的困難之一是每次僅能讀少量 byte 就發現要不到記憶體做 mapping 了，經過研究之後才想到必須要在 mmap 完後進行 `munmap` 才能釋出記憶體，如此一來，將傳輸能量從少量的 bytes 提升到 1GB 之多。 至於 `slave.c`，大致和 `master.c` 相同，只是身分從要寫資料進去的角色，變成讀資料的接收者。

`master_device.c / slave_device.c`: 這兩個檔案所使用的程式碼非常相似，因此將兩個檔案一起描述，並著重在`master_device.c`，但`slave_device.c`大同小異。為了支援`master.c`裡的mmap，我們在 master device 被 open 時就用 `kmalloc` 來要一塊作為 mmap 用途的 buffer。另外一個重要的因素是memory需要對齊才能正常執行，因此我們allocate memory時，都使用page size的倍數。 而在這部分，我們一開始觀察`dmesg`時會發現slave會不斷讀取到重複且不完整的片段。我們的解決方式是在`kreceive`的函式呼叫裡加上`MSG_WAITALL`的參數。這個參數會使得socket等待所有單次資料都傳輸完畢之後才會進行後續的操作，而不會在master的資料還沒傳送完畢之前就斷線。

`Testing` 為了測試不同page size/fcntl buffer size/mmap size與效能的關係時時，我們常常遇到程式crash的情況。經過觀察後我們發現sample code的 buffer 是開在 process memory 的 stack 區塊，所以當 buffer 一大，就會發生 stackoverflow的狀況。解決的方法是把變數放在 static 或是 global 區塊。

和 spec 比較不同的是，為了增加測試的效率，我們在 `user_program/master.c` 和 `user_program/slave.c` 裡面，我們使用 stdin 的方式來讀進參數，而不是透過 command line argument 來讀。

Bonus

我們發現在只要把 `struct socket` 這個資料結構中的 flags 這個變數加上 `FAYNC` (defined in `linux/fcntl.h`) 這個 flag，就能讓所有對這個 socket 的操作變成 asynchronous。所以我們只要在每一個 socket 被初始化時加上這個 flag，就能完成 bonus 要求的 asynchronous I/O. 在 sample input 2 上測試，我們發現平均傳輸時間降低了 5ms 左右，因此我們可以說 asynchronous I/O 是更有效率了。

Demo

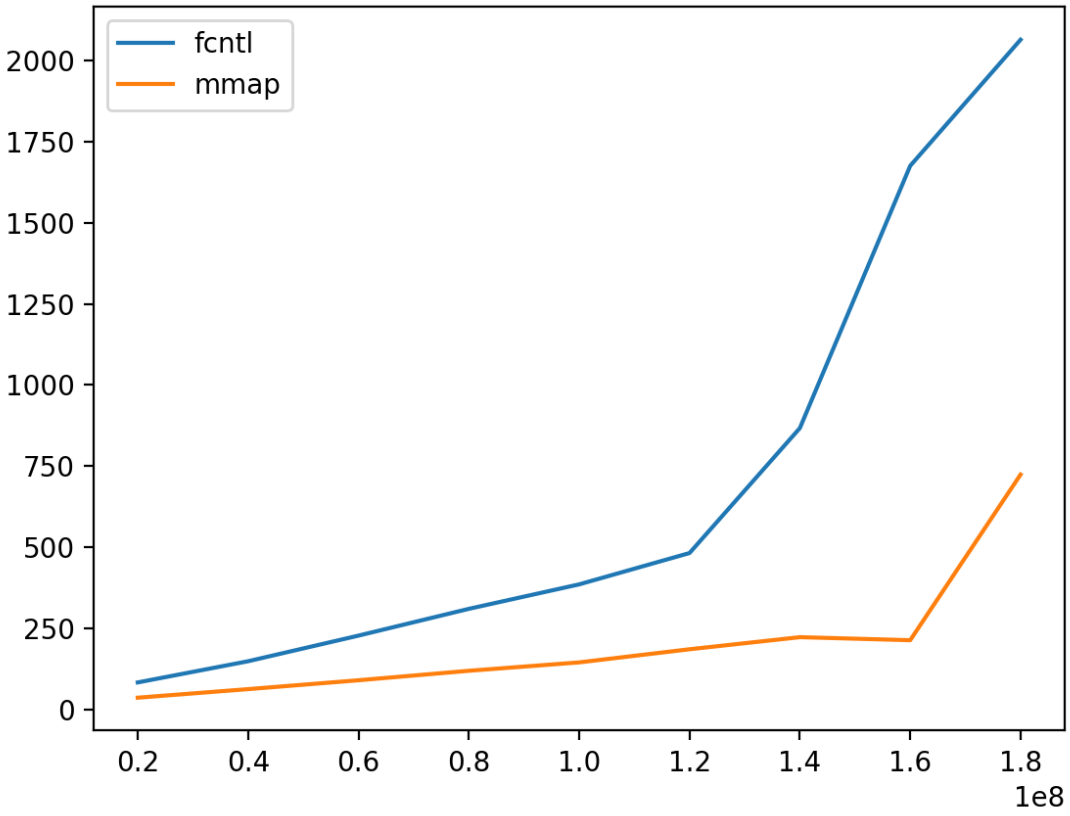
在 demo 的部分，我們對 3 種不同的 input 進行了測試，前 2 種爲助教提供的 sample input，分別代表多個小檔和單一大檔，而第 3 種爲我們自己設計的「多個大檔」(5個100MB大小的檔案)，我們認爲多個大檔比起單一大檔更能公平地去衡量不同傳輸方法的差異，避免因爲系統狀況的不穩定而產生有 bias 的結果。

下圖為 master device 與 slave device 的 page descriptor

```
[ 4579.668664] master: 800000006559067
[ 4579.669592] slave: 800000008132606 7
```

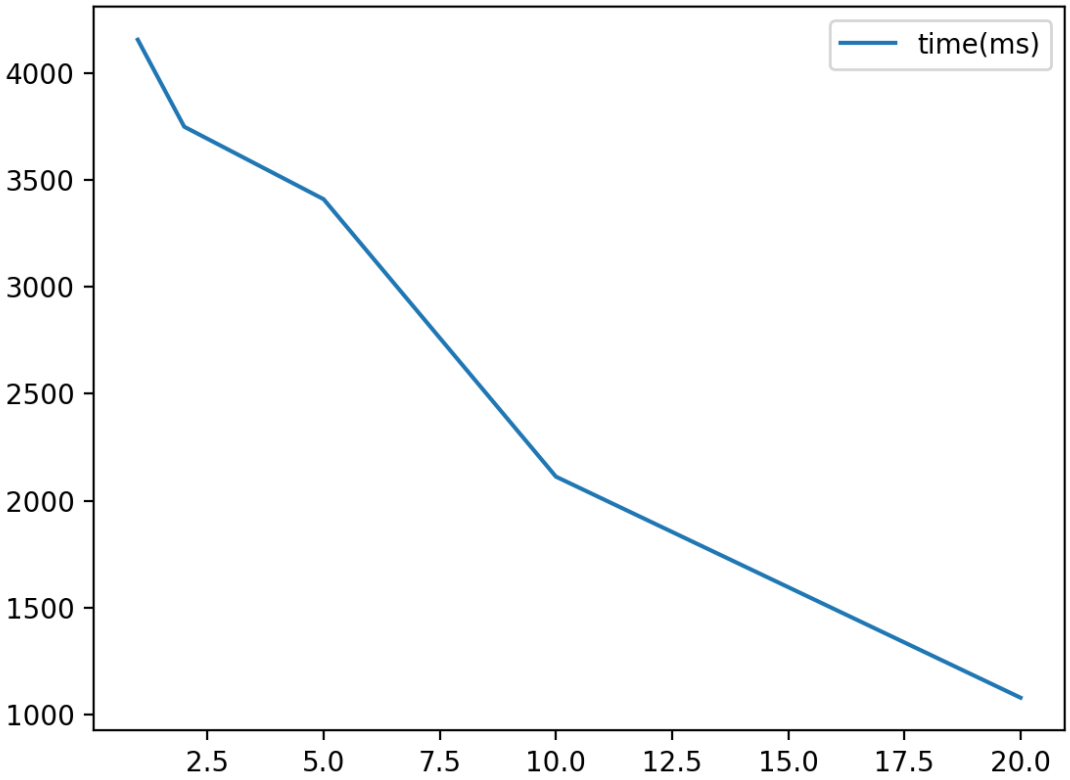
比較 read/write 和 mmap+memcpy

以下圖表為master與slave皆使用mmap與read/write，在不同file size時的表現：



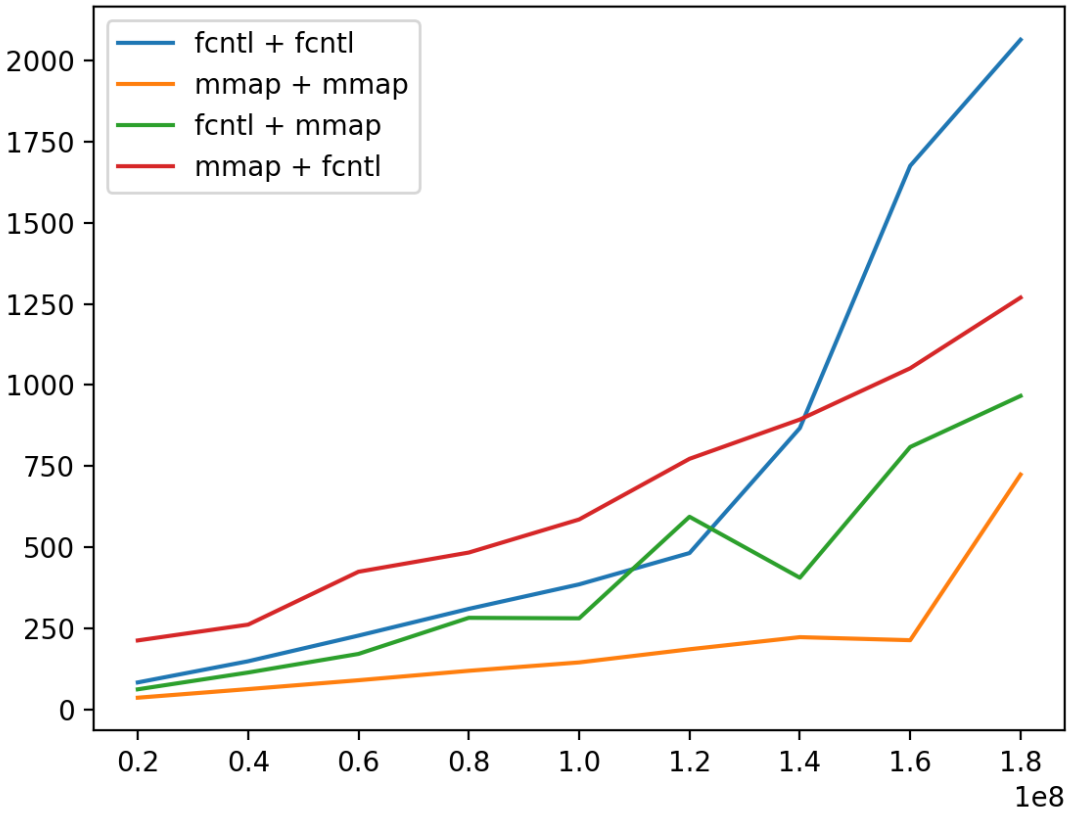
其中橫軸的單位為bytes。 從圖片中我們可以觀察到，mmap的表現普遍優於read/write，而且時間的差距隨著檔案的大小變大也更加明顯，應證了理論上mmap較優的表現。

下圖為使用mmap時，使用不同的mmap size(即單次傳輸的檔案大小)的表現。測試用的檔案大小為400MB。



其中橫軸的單位為page size的倍數，例如20即代表mmap size = $20\times$ page size。 可以觀察到隨著使用的mmap size增加，所需的時間也較少。原因是單次的傳輸量越高，所需要呼叫的mmap次數也降低，而單次傳輸量即使較高，在記憶體空間足夠的情況下，速度仍然是快速的。

下圖為master與slave分別使用不同的IO方式得出的時間表。格式$(X + Y)$代表master使用X，slave使用Y。



其中橫軸的單位為bytes，縱軸的單位ms。 圖中可以觀察到當兩邊都使用mmap時會得到最好的表現。而當兩方都使用fcntl時，在file size較大會有最差的表現。 另外我們認為slave的部分對整體表現的影響是比較大的。從圖中紅線與綠線可以觀察到，同樣是一方使用fcntl，一方使用mmap，但由slave使用mmap時表現會較優。

sample code 所做之修改

1. 裡面有很多的 `ioctl` 寫成了 `ioclt`
2. 以下這行 code 的 0.0001 應該要改成 0.001

```
    trans_time = (end.tv_sec - start.tv_sec)*1000 + (end.tv_usec -
start.tv_usec)*0.0001;
```

3. 以下這行 code 的 sizeof(msg) 應該要改成 count，這樣的程式邏輯才是 read 函數的程式邏輯。

```
    len = krecv(sockfd_cli, msg, sizeof(msg), 0);
```

4. 另外，最後在計算 transmission time 的時候，計算 byte 時應該不需要除以 8，會變成計算傳輸的 word 數。

分工表

學號 姓名	工作	分工占比
B07902028 林鶴哲	協助coding，並進行許多測試與debug，以及report的撰寫	16.66%
B07902064 蔡銘軒	修改原程式碼的錯誤，執行測試，寫report與製圖	16.66%
B07902075 林楷恩	撰寫基礎程式碼，執行測試，寫 report	16.7%
B07902084 鄭益昀	看 sample code，整體開發上的幫助，看一些 documentation 與對 mmap 的 debug	16.66%
B07902123 蔡奇夆	看 sample code，幫忙 kmalloc、kfree、receive_msg 的 debug，寫報告	16.66%
B07902136 楊子平	看 sample code，幫忙 mmap 的 debug、測試執行	16.66%

參考資料

- device driver 內的 mmap 實作： https://linux-kernel-labs.github.io/refs/heads/master/labs/memory_mapping.html https://github.com/paraka/mmap-kernel-transfer-data/blob/master/mmap-example.c `vma->vm_flags |= VM_LOCKED;` 來自於 B07902133。