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);  ioctl(dev\_fd, 0x12345676, (unsigned long)src); } |

## 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);  ioctl(dev\_fd, 0x12345676, (unsigned long)dst);  }  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); |

# 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 |

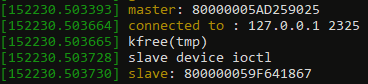
## page descriptors – fcntl(master) / mmap(slave)



## page descriptors – mmap(master) / fcntl(slave)



## page descriptors – mmap(master) / mmap(slave)



# 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 |
| **#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2×107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 |