2020 Operating System Project2

Group 23

B06902017　趙允祥
B06902105　吳吉加
B06902047　　陳彥
B06902111　林慶珠
B06902113　柯柏丞
B06902125　黃柏瑋

Summary

1 使用方式

1.1 環境架設

Linux Kernel Version : 4.14.25
由於input和output檔案過大，無法上傳github，因此需要經由 `./init.sh` 下載。

```
git clone https://github.com/joe0123/OS2020_Project2.git
cd OS2020-Project2-Group23/
sudo ./init.sh
sudo ./compile.sh
cd user_program/
```

1: 環境架設

1.2 使用說明

Master program 和slave program可以各自指定想要從device讀入的方式(File I/O 或Memory-mapped I/O)。

```
./master [N] [N filenames] [fcntl | mmap]
./slave [N] [N filenames] [fcntl | mmap] [IP of master]

./master 2 0.in 1.in mmap
./slave 2 0.out 1.out fcntl 192.168.1.1
```

2: 程式指令使用說明及示例

1.3 結果輸出

Standard Output Master program 和slave program 傳完檔案後會各自有如下的輸出在螢幕(stdout)。

```
Master: Transmission time: 1009.693800 ms, File size: 33554431 bytes
Slave: Transmission time: 3979.936300 ms, File size: 33554431 bytes
```

3: user program 輸出

Kernel Buffer Ring 我們把device的page descriptor寫在kernel buffer ring，檔案傳輸完畢可以使用 `dmesg` 查看裡面的訊息，如下所示。

```
[23579.391848] master: trying to print page descriptor...
[23579.391850] master: 8000000022E5F267
```

4: 使用說明及示例

```
[  622.234040] slave: trying to print page descriptor...
[  622.234041] slave: 8000000022022225
```

5: 使用說明及示例

2 設計

2.1 Memory-mapped IO & File IO 設計

Memory-mapped IO 我們Memory-mapped IO的實作方式是檔案每次切map size大小的資料（若不足則寫原本的資料大小），做map映射到user memory的某一塊，然後memcpy到另一塊記憶體（master是把file map的memory memcpy到device mapped memory；slave是把device的mapped memory memcpy到file map的memory），並且每次寫完map size大小的資料後做unmap，下一塊重新map。如 Figure 1。Memory-mapped IO的slave user program中，我們每一個檔案都開一個socket，檔案傳送完畢關閉這個socket；而device一開始就map，mapped-device在user memory的位置一直都是固定的，user program要結束時會再把device unmap，而kernel buffer ring列印的是master/slave device 的page descriptor。

File IO Master program的file IO會呼叫 `read()`，底層的system call會被呼叫處理interrupt。System call 會把要讀資料先放進kernel buffer，再return給user program，之後user program會呼叫 `write()`，master device中實作 `write()` 其實就是 `copy_from_user()` 和socket send。
Slave program則相反，呼叫 `read()` 把device的資料讀進user buffer（device做socket receive和 `copy_to_user()`），再對file呼叫 `write()`，system call處理interrupt把user buffer的東西複製到kernel buffer，再寫進disk裡。下圖中箭頭的說明文字是function，箭頭代表資料流向（受到function 操作）。如 Figure 2。

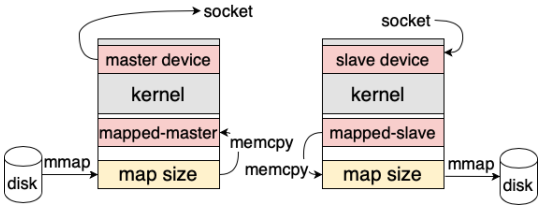

Figure 1: Memory-mapped IO

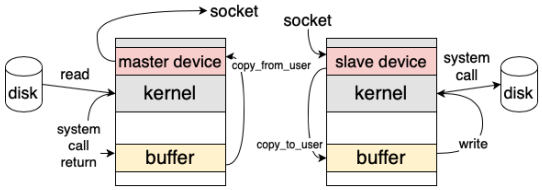

Figure 2: File IO

2.2 User Programs 和Devices的互動

兩個user program和各自devices的關係如Figure 3。用 `insmod` 載入driver時，master就把socket建立、 `bind`、 `listen`，直到slave user program open device後使用 `ioctl` 對device操作，會呼叫 `accept` 並等待slave連線。Slave device是在open後使用 `ioctl` 操作，包含建立socket和 `connect`。最後，我們的clean.sh 呼叫 `rmmod`，slave device會deregister，master device則會close socket以及deregister。

2.3 Bonus: Asynchronous Socket

我們透過原本synchronous的kernel socket改成asynchronous的版本，在socket的結構定義中，有一個flags屬性可以切換要使用的synchronous socket還是asynchronous socket，前者使用的flags為O_SYNC，後者使用的flags為FASYNC，意思說只要把每一個socket初始化的flags設為FASYNC即可達成socket的asynchronous。

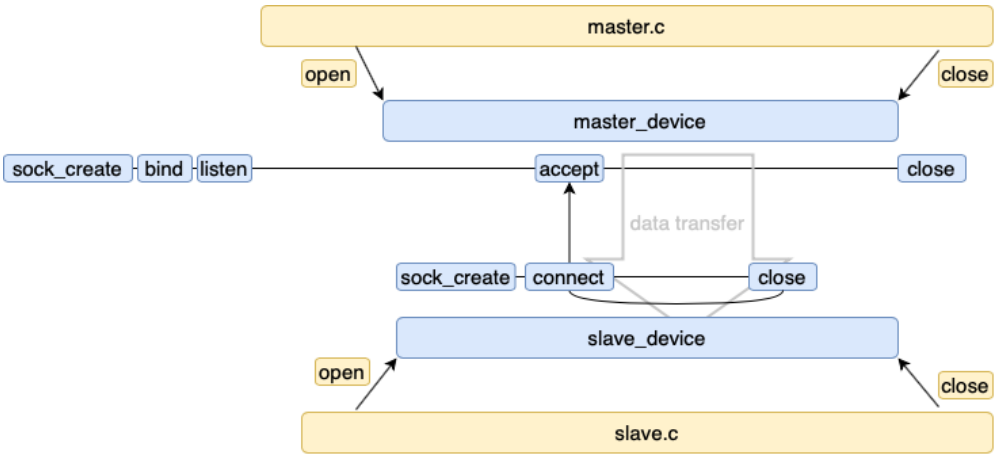

Figure 3: User programs和Devices的互動關係

3 效能

實驗設計動機 我們想知道Memory-mapped IO和File IO 這兩種方法有怎麼樣的差異，所以在Section 3.1 試圖比較File IO 和Memory-mapped IO的優劣，以及單檔傳送和多檔傳送的表現差異。Section 3.2 想要探討的是buffer/map size的增減會對檔案傳輸的效能有什麼影響。Section 3.3 要探討的是synchronous/asynchronous socket的表現。

實驗方法 我們把每個要測量的數據做了數十次實驗，拿掉距離平均最大的離群值，再取平均。會這麼做是因為我們發現數值浮動很大。我們使用的是一台real Notebook，不是用VM模擬，測量出來的數值較穩定。

3.1 File IO vs Memory-mapped IO

Buffer size和Map size相同 如果把buffer size設的比map size小，有可能Memory-mapped IO比較快是因為要執行的function次數較少，為了公平起見，這個section的實驗我們把**buffer size和map size設成一樣的值(4096 == PAGESIZE)**。

Sample Input Figure 4 是sample input的傳輸時間圖表。我們注意到sample input 1 的十個檔案加總起來才≈23.6 KB，而sample input 2 的大檔案是11MB。數據中顯示File IO比較快，尤其在sample input 2中，File IO甚至比Memory-mapped IO快上兩倍之多，但這和我們所學的知識有所違背，因此後來我們將input data加大並且統一總傳輸量後做了Generated Input這個實驗，發現使用Memory-mapped IO傳輸大檔案其實相當有效率。

Generated Input 因為我們覺得sample input的傳輸時間跑得很快，為了免除檔案太小造成誤差太大（可能光是開檔就佔了大部分時間等等）情形，我們把總共傳輸的資料量固定在1000MB，並分成1、5、10個檔案傳送。Figure 5 是master/slave 分別是Memory-mapped IO/File IO的4種組合。
（我們使用dd指令產生各種大小的檔案，例如：`dd if=/dev/urandom of=0.in bs=64M count=1`。）

1. 可以看到當slave傳送檔案的方式相同的時候，傳輸效能的表現比較相似（數值與增長趨勢等）。這可能是因為我們slave執行socket receive後會等到資料寫到disk完再呼叫下一個socket receive，而master的socket send會等到return再執行後面的指令。所以只有master使用Memory-mapped IO 對效能的提升沒有較大的幫助（還是需要等slave把該map size/buffer size寫進disk）；反之，slave使用Memory-mapped IO的幫助較為明顯（請看1個1000MB的檔案）。

2. Memory-mapped I/O在傳送大檔案時比File I/O還更效率，在Figure 5的1000MB的例子中，slave使用Memory-mapped I/O的速度搭約能比使用File I/O快上百分之二十，代表此時利用Memory-mapped I/O加速的效果已經超越map/unmap的overhead，為大檔的搬運增加不少效率。

3. 可以看到多個檔案的效能會比較差，因為我們的slave socket是每個檔案都create且connect一次，再加上每個檔案都需要開檔關檔，所以比起單一大檔效能較差。

4. Memory-mapped IO在5個檔案的情況下，可以看到slave是Memory-mapped IO的效能比較好。但到了10個100MB檔案的測試的時候4種傳輸方式組合已經非常接近，我們認為這時File IO的performance可能已經逼近甚至超過Memory-mapped IO，朝sample input的結果趨近。

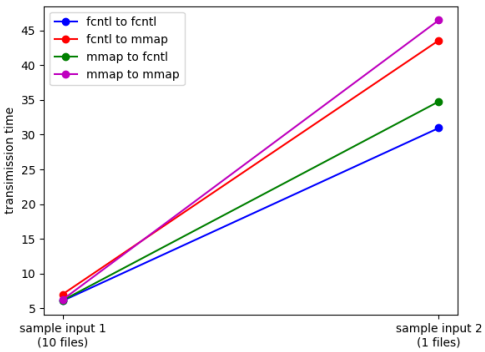

Figure 4: Sample Input

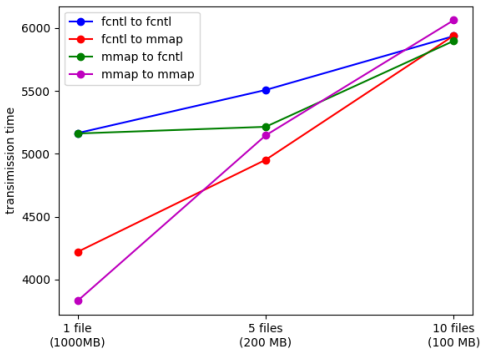

Figure 5: Generated Input

3.2 Buffer Size vs Map Size

從Figure 6 可以看到Memory-mapped IO的優勢：增加map size，因為一次搬較多東西，又可以減少map和unmap的次數，所以效率會比較好，搬運速度甚至可以加快三倍之多。同一張圖可以看出增加buffer size（需要開在global）可以增加效能，但增加的幅度不如Memory-mapped IO。Memory-mapped IO和File IO的效能增加幅度差異，我們猜測可能是mmap的overhead是在map和unmap指令（建立virtual mapping、page fault等等）；但File IO的bottleneck是在複製一份進kernel buffer。因為雖然減少了call function的次數，但複製進kernel buffer的資料沒有減少，所以提高的效能相當有限。

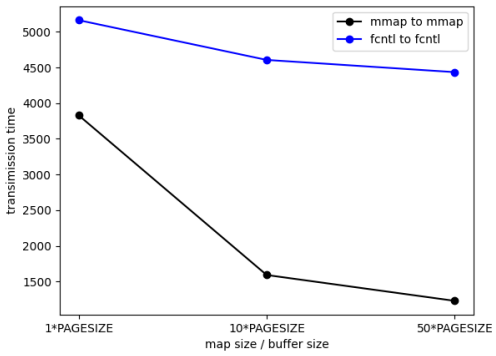

Figure 6: 不同buffer size和map size

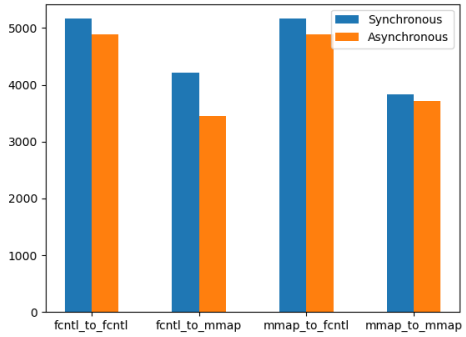

Figure 7: Sync和Async在兩種方法下的比較

3.3 Asynchronous vs Synchronous

Asynchronous的表現相對Synchronous優秀 從Figure 7 實驗數據的柱狀圖結果可得知，不管是在哪一種情境之下，Asynchronous都有花費比Synchronous更短的時間，推測是因為Asynchronous並不用像Synchronous每次都要等到buffer滿了才會把它清空，不過值得注意的是master端是使用Memory-mapped IO寫檔時，兩者之間的差距並沒有master端是使用File IO寫檔時來的大，猜測主要是跟buffer有關，其一是過小的buffer會導致兩者之間的差別縮小，其二是多次清空buffer所造成的成本也是Asynchronous必須承擔的。

3.4 小結

Memory-mapped IO和File IO的優劣 從Section 3.2 的實驗中可以得到：減少map/unmap 的次數可以明顯提升效能。這是因為map和unmap的overhead很大，因此，用Memory-mapped IO讀寫時如果過於頻繁地進行map/unmap，會大大降低讀寫的效能。
然而，Memory-mapped IO的優勢是可以把map size調得很大，傳大檔案的時候只要map/unmap次數少就可以很快傳完檔案。

依據檔案大小選擇傳輸方式 傳輸小檔案的時候File IO比較快，因為Memory-mapped IO的overhead太大（創建virtual mapping和page fault的penalty等等）（File IO不一定會有page fault，要看資料有沒有在page cache裡面）；如果是大檔案，就可以使用較大map size的Memory-mapped IO，對於效能提升應該會有顯著的效果。

4 討論

遇到的問題 在master以File I/O寫檔、slave以Memory-mapped I/O讀檔時，會遇到Memory-mapped I/O讀檔速度比File I/O寫檔快太多，導致報錯Segmentation Fault的問題。在將Memory-mapped端的mapping狀況印出來後，得到如下結果：
（左邊代表該輪的map size，右邊代表目標檔案目前的size）

```
4096 426283008
4096 426287104
4096 426291200
4096 426295296
4096 426299392
4043 426303488
53 426307531
./run_slave0.sh: line 1: 7704 Segmentation fault    sudo ./slave 1 0.out $1
    127.0.0.1
```

6: Memory-mapped端的mapping狀況

在我們使用的4.14.25 Linux Kernel 中，page size的大小是4096。根據上圖，當map size不到page size時，還不會出問題。但在下一輪map，當我們基於該輪socket接收到的資料量，讓file在對user space進行map時，要求超過53 bytes的空間，就會因為跨越page邊界而產生Segmentation Fault。因此，後來我們在socket接收資料的那端加上了waitall的flag，確定socket buffer滿了，才將資料放入device，就解決了這個問題。

穩定結果的方法 當我們在進行IO時，發現前幾次訪問的結果會相當糟糕，但連續多跑幾次後速度會越來越快，經過一些實驗和資料蒐集之後，我們發現這可能是因為page cache功勞，讓我們在讀寫文件時，可以用於緩存文件的內容，從而加快對disk的訪問。於是，如果我們在實驗之前執行以下指令，便可以大幅減少誤差，也讓我們更能聚焦在不同類型IO在讀寫上的差異。

```
sudo sh -c 'echo 3 > /proc/sys/vm/drop_caches'
```

7: 清除page cache的指令

5 組內分工表及分工比重

B06902017 趙允祥	devices	100%
B06902105 吳吉加	user programs	100%
B06902047 陳彥	user programs	100%
B06902111 林慶珠	report	100%
B06902113 柯柏丞	devices	100%
B06902125 黃柏瑋	devices	100%

6 Reference

1. Linux Kernel Document: 4.14.0 & 4.19.0
2. Memory-mapped IO: The GNU C Library
3. theoretically overhead analysis
4. implementation: Lu-YuCheng/OSProject2, domen111/OS-Porject-2