

OS Project2

Group 38

Github Link: <https://github.com/edisonhello/OS-Project2>

Members and Work Distribution

資工二 b07902024 塗大為

- 實作 user_program/slave 以及 slave_device 中 mmap 部份。
- 將 fcntl 與 mmap 兩者套用我們設計的 protocol。
- 報告撰寫。

資工二 b07902131 陳冠宇

- 實作 user_program/master 以及 master_device 中 mmap 部份。
- 將 fcntl 與 mmap 兩者套用我們設計的 protocol。
- 報告撰寫。

資工二 b07902132 陳威翰

- demo 影片錄製。
- 繪製圖表。
- 報告撰寫。
- 程式除錯。
- 設計測試腳本與測試檔案。

資工二 b07902133 彭道耘

- demo 影片錄製。
- 繪製圖表。
- 報告撰寫。
- 程式除錯。
- 設計測試腳本與測試檔案。

資工二 b07902143 陳正康

- 繪製圖表。
- 報告撰寫。
- 程式除錯。
- 設計測試腳本與測試檔案。

Design

- Protocol: 由於需要傳輸多個檔案，我們不能將所有的檔案的內容全部皆在一起傳出去，因此我們需要設計一個 Protocol。我們最終決定使用的 Protocol 非常直觀：master 在傳輸檔案的時候會先將檔案的大小傳給 slave，相對的 slave 在接收檔案前會先讀入檔案大小的資訊，其中檔案大小用一個 8-byte 的 unsigned 整數來表示，而所有要傳送的檔案就依照這個模式，每個分開紀錄大小，再將要傳送的 bytes 串接再一起，一併送出。例如當我們要先傳送一個大小為 7 bytes 的檔案再傳送一個大小為 17 bytes 的檔案時，以下的 byte sequence 會被送出與接收：

```
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x07 0x01 0x02 0x03 0x04 0x05 0x06 0x07
0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x11 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF
0xFF
0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF
```

- User Programs: 為了方便解釋，以下以只傳送與接收一個檔案為例：
 - fcntl: 透過 struct file_operations {master,slave}_fops 中指定這個 device 被 read 跟 write 時分別要執行哪個函式，改寫其行為為透過 ksocket 傳送 / 接收。接著 user 端只需要以正常從檔案中讀取跟寫入的方式便可以控制 device 進行檔案的收發。
 - mmap: 傳送與接收資料的流程為：master 先從 master_device 中 mmap 出一個 page，接著將要傳送的資料利用 memcpy 複製到該 page 中，最後呼叫對 master_device 呼叫 ioctl 將資料送出。slave 的部分也大同小異：首先從 slave_device 中 mmap 出一個 page，對 slave_device 呼叫 ioctl 接受資料，最後 memcpy 出該 page 中的資料寫到目的檔案中。
- Devices
 - 當 master_device 跟 slave_device 完成連線後，兩者皆會使用 kmalloc(2 * PAGE_SIZE, GFP_KERNEL) 在 kernel 裡 allocate 一塊大小為兩倍 page size 的連續記憶體。將大小設為兩倍 page size 的用意是這樣我們就保證一定可以在這 2 * PAGE_SIZE 中找到連續 PAGE_SIZE 的 bytes，且開頭是 PAGE_SIZE 的倍數（即完整的一個 page）。這段 physically contiguous 的記憶體以 kernel_mem 來代稱。
 - 首先在 struct file_operations {master,slave}_fops 中將 mmap 這個 field 指定到我們實作的 {master,slave}_mmap 函式中，這會告訴 kernel 當 {master,slave}_device 被呼叫 mmap 這個 system call 時，要呼叫 {master,slave}_mmap 這個函式，如此一來我們便能在該函式中將 user space 中的記憶體與 kernel space 做對應。{master,slave}_mmap 會傳入兩個參數 struct file *filp 以及 struct vm_area_struct *vma，前者代表這個 device 被 open 所對應的 open file 的資訊，而後者則代表 user 想要 mmap 區域的相關資訊，其中 vma->vm_start 跟 vma->vm_end 代表使用者想要將 user virtual address space 中的 vma->vm_start 到 vma->vm_end 對應到某段 physical address，而在這次的 project 中對應的就是 device 在最先開始 allocate 出來的 kernel_mem。由於使用者端也是又我們實作，因此能保證 vma->vm_end - vma->vm_start 剛好就是一個 page 的大小，也就能剛好對應上。

- 因此，當 device 被呼叫 `mmap` 以後，我們先將 `vma->vm_flags` 加上 `VM_LOCKED`，如此一來這個 page 就不會被 swap out。接著使用 `remap_pfn_range` 這個函式將 `[vma->vm_start, vma->vm_end)` 對應到 `kernel_mem` 上，要注意的是 `kernel_mem` 是由 `kmalloc` 得到的 address，而 `kmalloc` 給的是 `kernel space` 中的 *virtual address*，而 `remap_pfn_range` 又需要被對應的 address 為 *physical address*，因此我們必須先透過 `virt_to_phys` 將 *virtual address* 轉為 *physical address*（註1：由於 `kernel virtual address space` 分成兩個部分，一個與 `kernal physical address space` 有線性的——對應（差一個 offset）而另一個則需透過 `page table` 作轉換，而 `kmalloc` 出來的記憶體會位於第一個部分，因此 `virt_to_phys` 實際上只是將傳入的 address 扣掉一個常數而已）（註2：與 `kmalloc` 相對的另一個函式為 `vmalloc`，這個函式會 allocate 第二個部分的記憶體，因此 `vmalloc` 所回傳的記憶體只保證在 *virtual address space* 上連續，並不能保證其對應 *physical address space* 上的連續性）。
- `remap_pfn_range` 的 prototype 如下：`int remap_pfn_range(struct vm_area_struct * vma, unsigned long addr, unsigned long pfn, unsigned long size, pgprot_t prot);` 而這個函式做的事情就是在該 process 的 `page table` 上加上 `addr` 到第 `pfn` 個 *physical page*（`kernel_mem` 剛好是 `PAGE_SIZE` 的倍數，因此我們可以直接傳入 `kernel_mem >> PAGE_SHIFT`）長度為 `size` 的這個對應，並且會在這些 `page` 上加上 `prot` 所指定的保護機制。在呼叫完 `remap_pfn_range` 了以後，任何在 `kernel_mem` 中的記憶體都可以在 `user space` 中透過 `mmap` 回傳的 `pointer` 來 access。
- `mmap` 完畢以後，以 `master_device` 為例，利用 `ioctl` 指定的 `ioctl_num` 為 `master_IOCTL_MMAP`（即 `0x12345678`）時的第三個參數作為此次要實際傳送的 `byte` 數，呼叫 `send_mmap` 之後，使用 `ksend` 將在前面一點提到的記憶體空間中，`user_program` 複製進來的資料實際透過 `socket` 送出給 `client` 端。相對的，`slave_device` 就是在 `user_program` 指定完要接收的 `byte` 數量之後，藉由 `krecv` 接收資料內容，並將資料放到 `mmap` 的記憶體中，讓 `user_program` 能夠從 `mmap` 出來的空間將其複製出來。
- Page Descriptor:
 - 當每次傳送或接收完一個檔案之後，使用者需要將之前 `mmap` 出來的 `memory` `unmap` 掉，在這之前我們會呼叫 `ioctl(dev_fd, 0, ptr)`，其中 `dev_fd` 為 device 所對應的 `file descriptor` 而 `ptr` 為 `mmap` 出來的 `pointer`。在 device 中的 `{master,slave}ioctl` 函式中，若 `IO control` 的指令並不是特定的指令（連線、`mmap` 等），則會透過在一層一層的 `page table` 中走來得到傳入參數最對應到的 *physical address*。
 - 由於我們使用的是版本較新的 `Linux Kernel (4.15.0)`，`Linux` 中所使用的 `page table` 已經由原本的四層設計改為五層的設計（在 `version 4.11-rc2` 時加入），也就是從原先的

PGD -> PUD -> PMD -> PTE

的架構中加入一層 `P4D`，改為

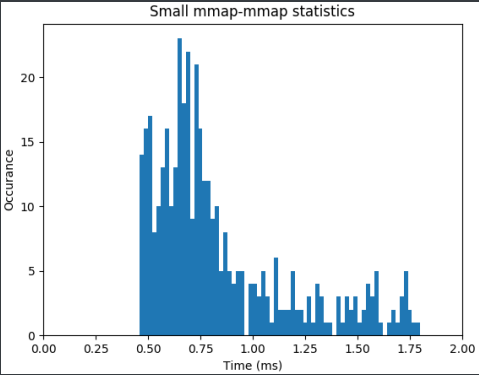
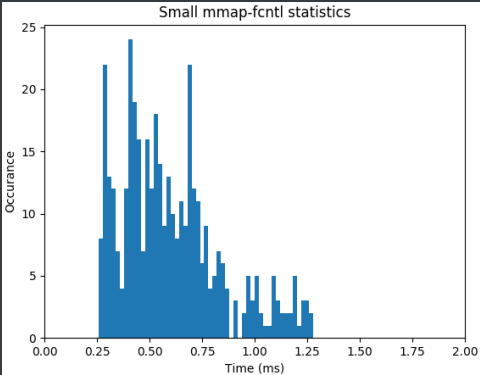
PGD -> P4D -> PUD -> PMD -> PTE

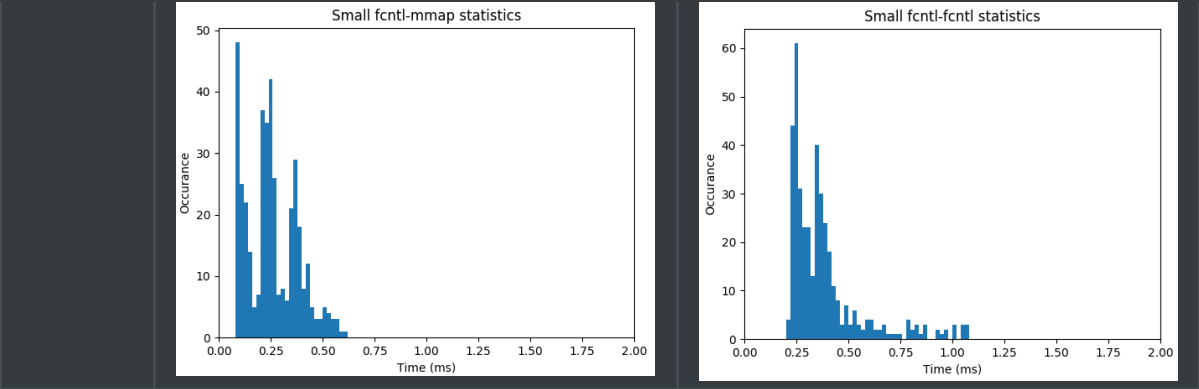
這個改動主要是為了因應未來機器所擁有 memory 會越來越多的趨勢。雖然原本的四層架構可以應付現行 x86_64 中 48-bit 的 virtual address space，增加的五層架構可以在之後 x86 支援 52-bit physical address space 與 57-bit virtual address space 時達到比較好的效能。獲得 page table entry 的方式也非常直覺：透過 pgd_offset 、 p4d_offset 、 pud_offset 、 pmd_offset 與 ptf_offset_kernel 等函式一層一層獲得各層的 handle，最後即可得到真正的 physical address。

- Input:
 - input/input_3 此為自己準備的測試檔，裡面一共有三個檔案，總大小為 1.1 MB：
 - target_file_v3_1
 - target_file_v3_2
 - target_file_v3_3
 - 這些檔案的內容是由隨機的整數（範圍 $[0, 10^9]$ ）所組成，隨機內容的好處為這樣程式的錯誤比較容易被抓到，比起一些有重複 pattern 的檔案，傳送端或接收端必須要完全正確的執行才能在隨機的檔案上得到正確的結果，這樣也就避免了例如有一個 page 被重複傳送或接受的錯誤行為。

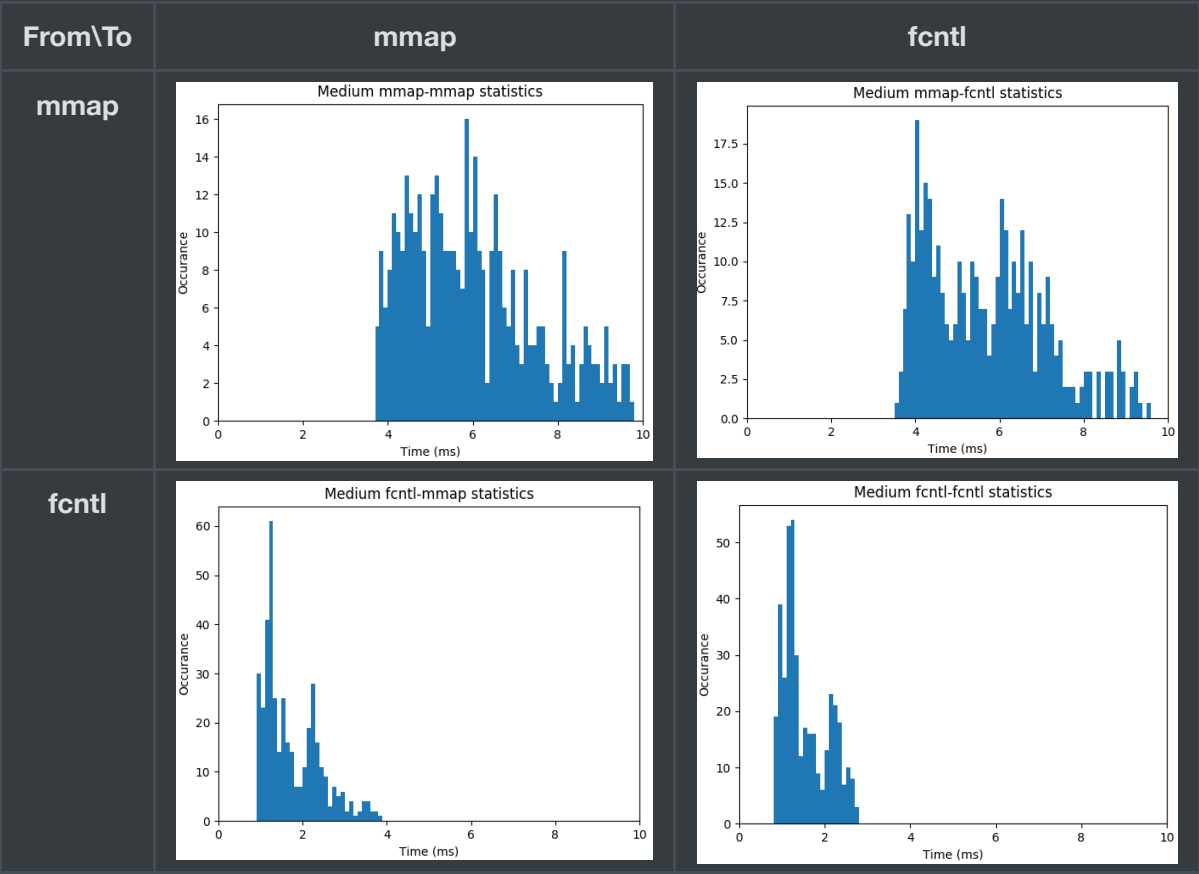
Result

- 測試環境：Virtualized Ubuntu 16.04, linux-4.14.25 with 3 CPU cores and 3GB of RAM.
- Page descriptors:
 - mmap to mmap: **[115.420404] slave: 80000001174CE225**
[115.420309] master: 8000000119736267
- 我們進行了分別測試了三組不同的資料，蒐集了 500 筆資料後去除極端值得結果如下：
 - Small Data (Total 48 KB, 10 files):

From\To	mmap	fcntl
mmap		
fcntl		

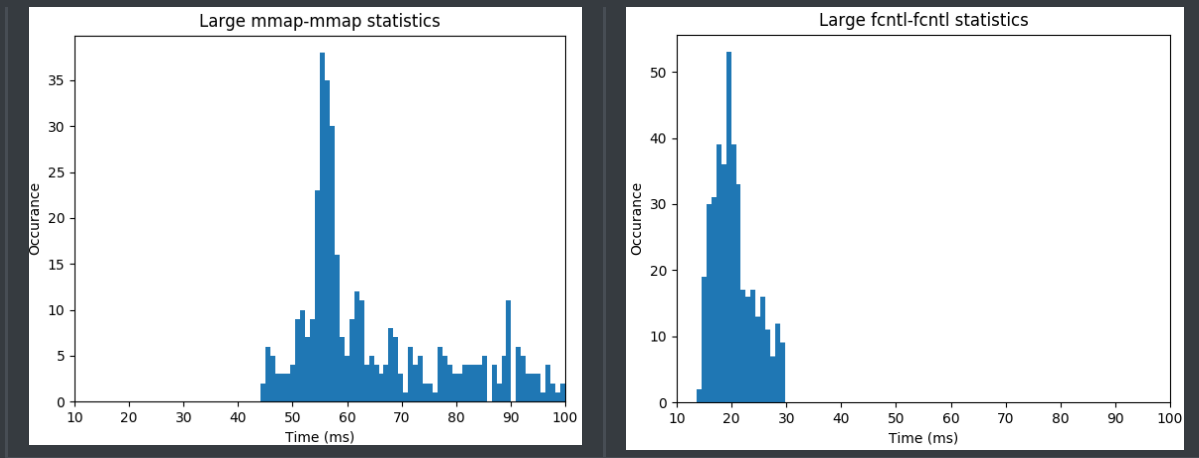


- Medium Data (Total 1.1 MB, 3 files):



- Large Data (Total 12 MB, 1 files):

mmap to mmap	fcntl to fcntl



- File I/O v.s. Memory-mapped I/O
 - 由於 mmap 可以將資料載入位於 kernel space 的 cache buffer 後，把地址映射到 user space 的 virtual address，不需要再次經過一次複製到 user space 就可以進行讀寫，因此理論上可以加速 I/O 的效率。
 - 然而以實際 demo 結果來看的話，普遍來講 fcntl 的速度是快於 mmap 的。因此我們在不同機器分別測試了 from mmap to mmap 跟 from fcntl to fcntl 兩者的速度比較，發現在不同機器上的結果會有顯著的不同。推測有可能是因為 VM 以及 host machine 上架構的不同所影響。
 - 另一個可能造成 mmap 沒有想像中表現那麼好的可能原因是 mmap 的大小造成，由於我們不管在由 kernel 讀寫資料或是將資料讀寫至檔案時，都是一次以一個 page 來進行的，如此頻繁的呼叫 mmap/munmap 可能會造成一些不必要的 overhead，進而影響整體的效能。

References

- Linux Memory Mapping (https://www.cs.binghamton.edu/~ghose/CS529/sharedmemmap.htm?fbclid=IwAR2iLyI241b5iBRLpd5UApctiYxpRyUocQ_mXUYRBghTZw5-F-bq1cHeN-4)
- The mmap Device Operation (https://www.oreilly.com/library/view/linux-device-drivers/0596000081/ch13s02.html?fbclid=IwAR0j0JQ3IbD3Na9-uBogJq_37JT6nXnU96LglcNjcZEOK7ULZWTERmbzKms)
- Char drivers (chapter 3), Linux Device Drivers, Third Edition (<https://lwn.net/Kernel/LDD3/>)
- Linux Kernel Documentation (<https://www.kernel.org/doc/html/latest/index.html>)
- Process Address Space (<https://www.kernel.org/doc/gorman/html/understand/understand021.html>)
- Linux Kernel Source code for memory management (<https://elixir.bootlin.com/linux/v4.15/source/mm>)
- Page Table Management (<https://www.kernel.org/doc/gorman/html/understand/understand006.html>)
- Five Level Page Table (<https://lwn.net/Articles/717293/>)