Operating System Project 2 Report

設計

slave.c

- 因為會有需要讀好幾個檔案的情形，我們先將整段程式碼放在一個 `for` 迴圈內，每次讀取一個檔案的內容就會開啟 socket，當該檔案讀完之後再關閉 socket，直到所有檔案的讀取都完成。
- 內部則修改了 `mmap` 的 case，每次迴圈都會將 `offs` 初始化為0，並設定 `r` 為讀進來的訊息的大小，如果已經沒有內容要讀進來就讓檔案大小 `file_size` 加上 `offs` 的大小並跳出迴圈，否則繼續讀資料。
- 我們使用 `posix_fallocate` 這個函式來確保 disk 的 space 從 `offs` 開始往後 `r` 個 byte都能夠被 allocate。
- 運用 `mmap` 這個函式將 calling process 映射到 virtual address，`mmap(NULL, r, PORT_WRITE, MAP_SHARED, file_fd[i], offs)` 的意思是以 page-aligned 的方式 map 長度 `r` 的 address，在 `file_fd` 中以 write 的方式映射，而`mmap(NULL, r, PORT_READ, MAP_SHARED, file_fd[i], offs)` 也是類似，只是在 `dev_fd` 中是以 read 的方式映射，而 `MAP_SHARED` 的意思是如果有任何一個process 有更動 mapping ，其他同一個區域的 process 也可以看到。
- `memcpy`是從 `kernel_add` 中 copy `r` 個 bytes 到 `file_add`中，另外，每次讀資料後 `offs` 都會增加 `r` 的大小。

```
for (int i = 0; i < num; i++) {
    if(ioctl(dev_fd, 0x12345677, ip) == -1){        //0x12345677 : connect to master in the device
        perror("ioclt create slave socket error\n");
        return 1;
    }

    //write(1, "ioctl success\n", 14);

    switch(method[0])
    {
        case 'f'://fcntl : read()/write()
        do {
            r = read(dev_fd, buf, sizeof(buf)); // read from the the device
            write(file_fd[i], buf, r); //write to the input file
            file_size += r;
        }while(r > 0);
            break;
        case 'm'://mmap
            offs = 0;
            int flag = 1;
            while (flag == 1) {
                r = ioctl(dev_fd, 0x12345678);
                if (r == 0) {
                    file_size += offs;
                    flag = 0;
                    break;
                }
                posix_fallocate(file_fd[i], offs, r);
                file_add = mmap(NULL, r, PROT_WRITE, MAP_SHARED, file_fd[i], offs);
                kernel_add = mmap(NULL, r, PROT_READ, MAP_SHARED, dev_fd, offs);
                memcpy(file_add, kernel_add, r);
                munmap(file_add, r);
                munmap(kernel_add, r);
                offs = offs + r;
            }
            ioctl(dev_fd, 0x12345680); //default
            break;
    }

    if(ioctl(dev_fd, 0x12345679) == -1) {// end receiving data, close the connection
        perror("ioclt client exits error\n");
        return 1;
    }
}
```

---

master.c

- 如果有多個 input 檔案，每次傳完整個檔案之後，關掉 socket，之後再重開，以確保接收端可以知道每個檔案的開始與結束
- 利用 `mmap()` 將 input file 的內容 map 到一塊記憶體(file_map)，device 的部分也用 `mmap()` 得到一塊記憶體(device_map)之後，將 file_map 的內容複製到 device_map，其中每次 `mmap()` 的大小為一個 `PAGE_SIZE`
- 再呼叫 `ioctl()` 使 master device 執行 mmap 相應的工作
- 最後呼叫 `ioctl()` 讓 master device 執行 default，將 device 的所占用的 physical memory 的 adress 印出

```
case 'm'://mmap
    offs = 0;
    int flag = 1;
    while (flag == 1) {
        r = ioctl(dev_fd, 0x12345678);
        if (r == 0) {
            file_size += offs;
            flag = 0;
            break;
        }
        posix_fallocate(file_fd[i], offs, r);
        file_add = mmap(NULL, r, PROT_WRITE, MAP_SHARED, file_fd[i], offs);
        kernel_add = mmap(NULL, r, PROT_READ, MAP_SHARED, dev_fd, offs);
        memcpy(file_add, kernel_add, r);
        munmap(file_add, r);
        munmap(kernel_add, r);
        offs = offs + r;
    }
    ioctl(dev_fd, 0x12345680); //default
    break;
```

---

slave_device.c

- 先定義device被mmap呼叫時用來當handler的函式`slave_mmap`，其中
  - 用`virt_to_phys()`函式得到file的physical address後，右移`PAGE_SHIFT`算出page offset
  - `io_remap_pfn_range()`將vm *vma map到user space
  - 調整一些參數
    - `VM_RESERVED`避免vm被swap out
    - vm options 及 private data的更改

```
static int slave_mmap(struct file *file, struct vm_area_struct *vma){
  vma->vm_pgoff = virt_to_phys((void *)file->private_data)>>PAGE_SHIFT;
  if(io_remap_pfn_range(vma, vma->vm_start, vma->vm_pgoff, vma->vm_end - vma->vm_start, vma->vm_page_prot)){
    printk("pfn failed\n");
    return -1;
  }
  vma->vm_ops = &slave_vmops;
  vma->vm_flags |= VM_RESERVED;
        vma->vm_private_data = file->private_data;
  vm_open(vma);
  return 0;
}
//qq
//file operations
static struct file_operations slave_fops = {
        .owner = THIS_MODULE,
        .unlocked_ioctl = slave_ioctl,
        .open = slave_open,
        .read = receive_msg,
        .release = slave_close,
        .mmap = slave_mmap
};
```

- 接著當slave呼叫mmap時，device呼叫ksocket.c中有的函式 `krecv`，使slave能從socket接收file，並得到回傳值，也就是該file的長度。

```
case slave_IOCTL_MMAP:
        ret = krecv(sockfd_cli, file->private_data, MAP_SIZE, 0);
        break;
```

---

master_device.c

- mmap handler slave_mmap 與 master_mmap 相同

```
static int master_mmap(struct file *file, struct vm_area_struct *vma){
  vma->vm_pgoff = virt_to_phys(file->private_data) >> PAGE_SHIFT;
  if(io_remap_pfn_range(vma, vma->vm_start, vma->vm_pgoff, vma->vm_end - vma->vm_start, vma->vm_page_prot)){
    printk("pfn failed\n");
    return -1;
  }
  vma->vm_ops = &master_vmops;
  vma->vm_flags |= VM_RESERVED;
        vma->vm_private_data = file->private_data;
  vm_open(vma);
  return 0;
}
//qq

//file operations
static struct file_operations master_fops = {
  .owner = THIS_MODULE,
        .unlocked_ioctl = master_ioctl,
        .open = master_open,
        .write = send_msg,
        .release = master_close,
        .mmap = master_mmap
};
```

- 當master呼叫mmap時，device呼叫ksocket.c中的函式 `ksend`，使master能透過socket傳送

file，並得到回傳值，也就是該file的長度。

```
 case master_IOCTL_MMAP:
                //qq
                ret = ksend(sockfd_cli, file->private_data, ioctl_param, 0);
                //qq
          break;
```

Bonus

1. asynchronous I/O

```
ksocket_t ksocket(int domain, int type, int protocol)
{
        struct socket *sk = NULL;
        int ret = 0;

        ret = sock_create(domain, type, protocol, &sk);
        if (ret < 0)
        {
                printk(KERN_INFO "sock_create failed\n");
                return NULL;
        }
#ifdef ASYNC
    sk->flags |= FASYNC;
#endif
        /*
        if (sk && sk->sk) {
                if (sk->sk->sk_data_ready) {
                        origSk = sk->sk->sk_data_ready;
                        sk->sk->sk_data_ready = yh_sk_data_ready;
                } else {
                        printk(KERN_INFO "sk->sk->sk_data_ready is NULL\n");
                }
        } else {
                printk(KERN_INFO "sk or sk->sk is NULL\n");
        }
        */

        printk("sock_create sk= 0x%p\n", sk);

        return sk;
}
```

```
ksocket_t kaccept(ksocket_t socket, struct sockaddr *address, int *address_len)
{
        struct socket *sk;
        struct socket *new_sk = NULL;
        int ret;

        sk = (struct socket *)socket;

        printk("family = %d, type = %d, protocol = %d\n",
                                        sk->sk->sk_family, sk->type, sk->sk->sk_protocol);
        //new_sk = sock_alloc();
        //sock_alloc() is not exported, so i use sock_create() instead
        ret = sock_create(sk->sk->sk_family, sk->type, sk->sk->sk_protocol, &new_sk);
        if (ret < 0)
                return NULL;
        if (!new_sk)
                return NULL;
#ifdef ASYNC
    sk->flags |= FASYNC;
#endif

        new_sk->type = sk->type;
        new_sk->ops = sk->ops;
```

- socket 可以選擇 sync 或 async，我們將 socket 的 flag 更改為 對應的 `FASYNC`

2. design input parameter

```
./master file_num input_filename method
./slave file_num output_filename method ip
```

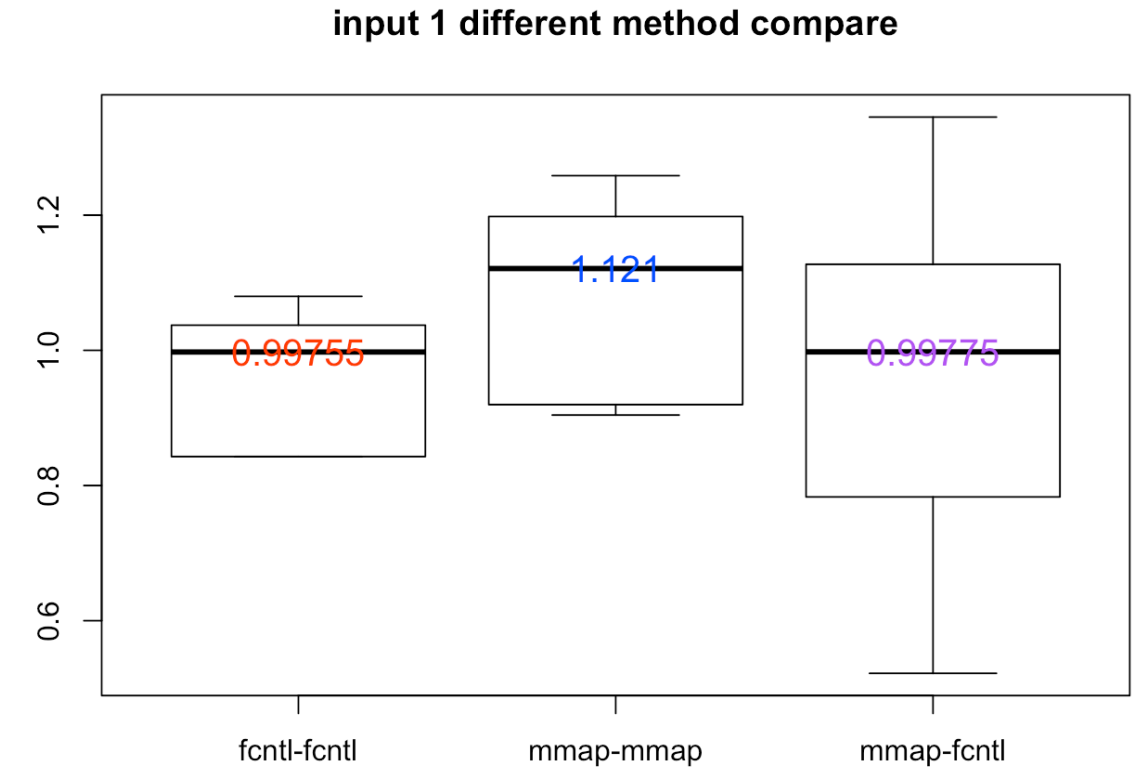
- 檔案名稱格式
  - 規定 input files 都為 "{input_filename}_{i}" 格式
  - 規定 output files 都為 "{output_filename}_{i}" 格式
- 在 `master.c` 和 `slave.c` 依照參數中輸入的檔案數量及檔案開頭，生成所有檔案名稱

比較 file I/O 和 memory-mapped I/O 的結果與效能差異並解釋

不同 input 針對不同 method 的比較

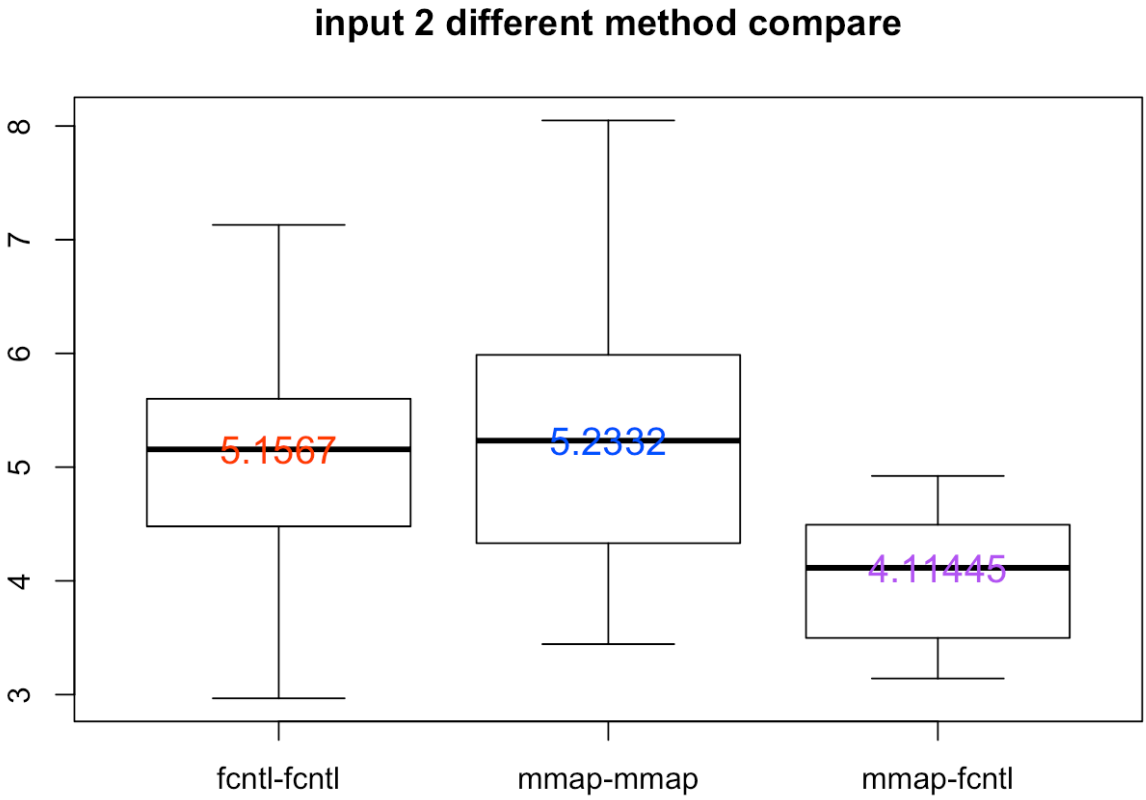
除了題目要求的 fcntl-fcntl 與 mmap-mmap 的比較，我們另外加入的 mmap-fcntl 繪圖比較

- sample input 1 的 fcntl-fcntl、mmap-mmap、mmap-fcntl 的比較



這是經由多次實驗後畫出的盒方圖，檔案大小為3018 bytes，可以發現在傳送多個小檔案時，fcntl-fcntl 的表現會較 mmap-mmap 好，推測可能的原因是 mmap 在實作時，會需要動態更動 page table 內容等額外的 overhead，另外在 CPU 送出一個位置時會先去查找 TLB，若 TLB miss 會再去看 page table中是否為 valid，若為 invalid 代表有 page fault，如果發生 page fault 且使用 mmap 反而會花費較多時間。

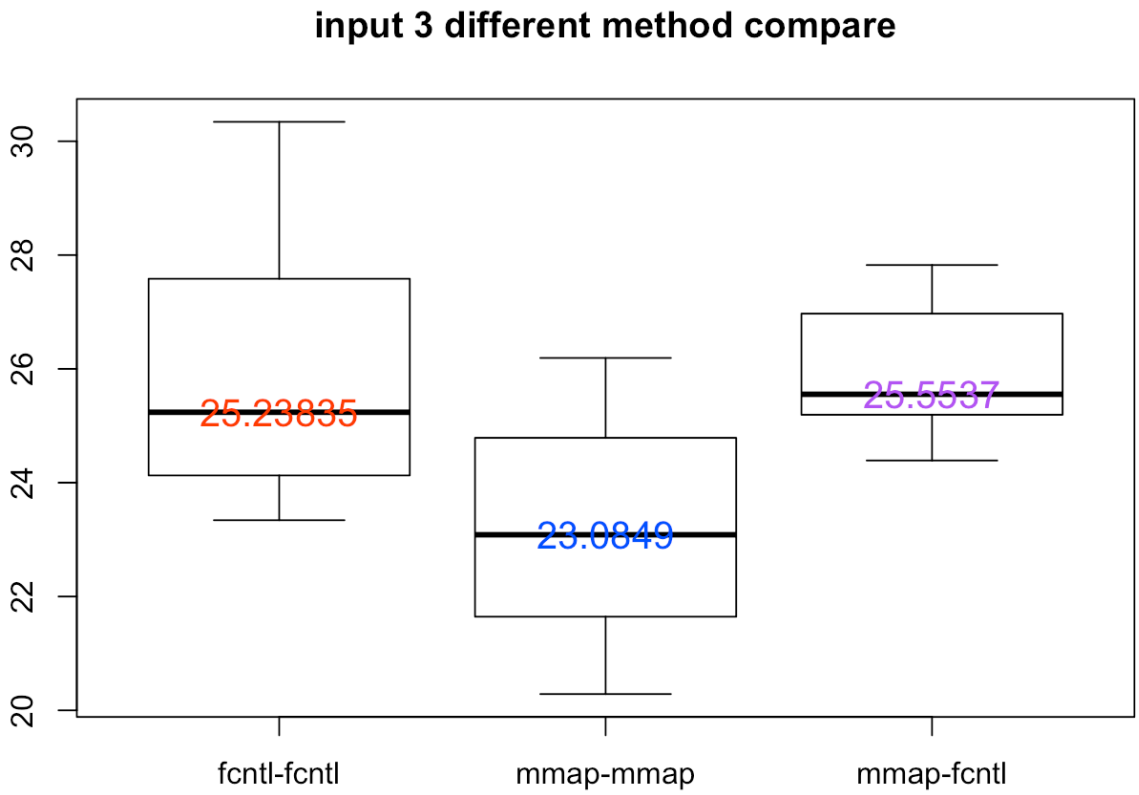
- sample input 2 的 fcntl-fcntl、mmap-mmap、mmap-fcntl 的比較



這是經由多次實驗後畫出的盒方圖，檔案大小為1502860 bytes，可以發現在傳送一個稍微較大的檔案時，mmap-mmap 的效能會較 fcntl-fcntl 的表現好一些，推測原因為較少的 disk I/O 等待時

間。

- our input 的 fcntl-fcntl、mmap-mmap、mmap-fcntl 的比較

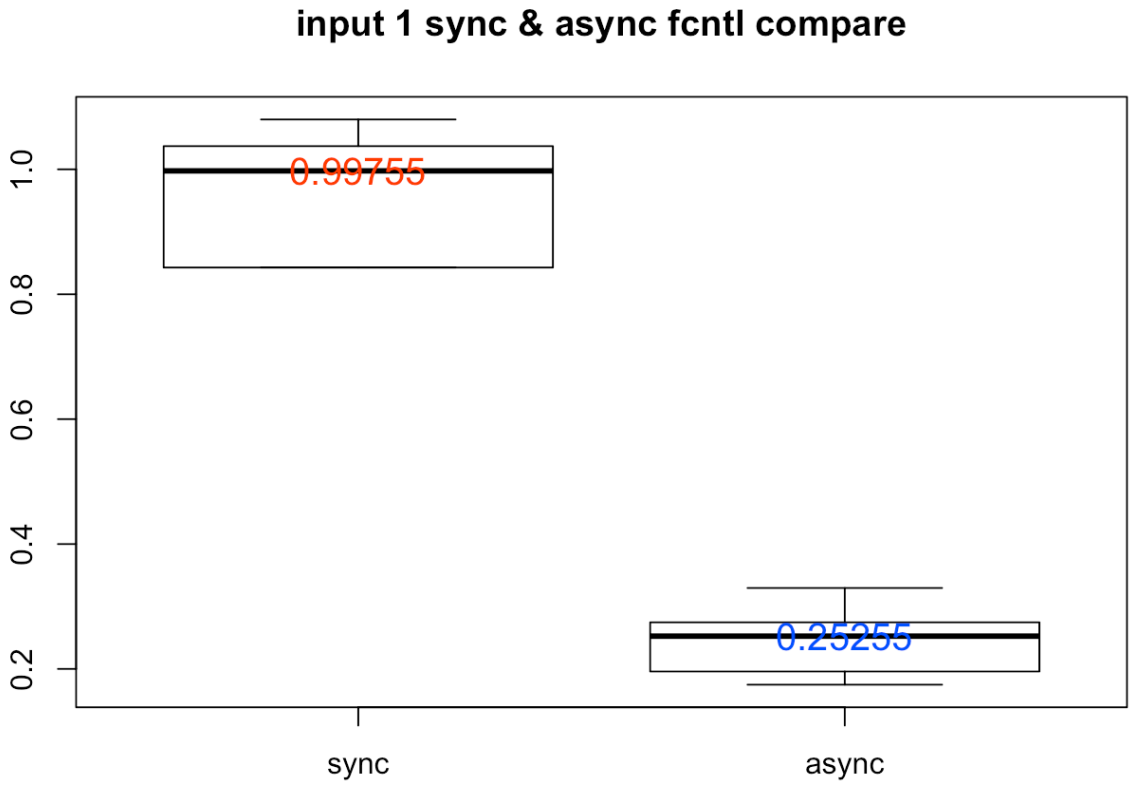


這是經由多次實驗後畫出的盒方圖，input 的內容為我們自己產生的測資，總共有5個檔案，總共的檔案大小為9017255 bytes，可以發現隨著檔案大小的增加，mmap-mmap 的效能較 fcntl-fcntl 的效能好的情形更加顯著，推測原因為每次 `mmap` 的使用為 memory 的 copy，不用每次都要完成 disk I/O ，因此可以節省許多時間。

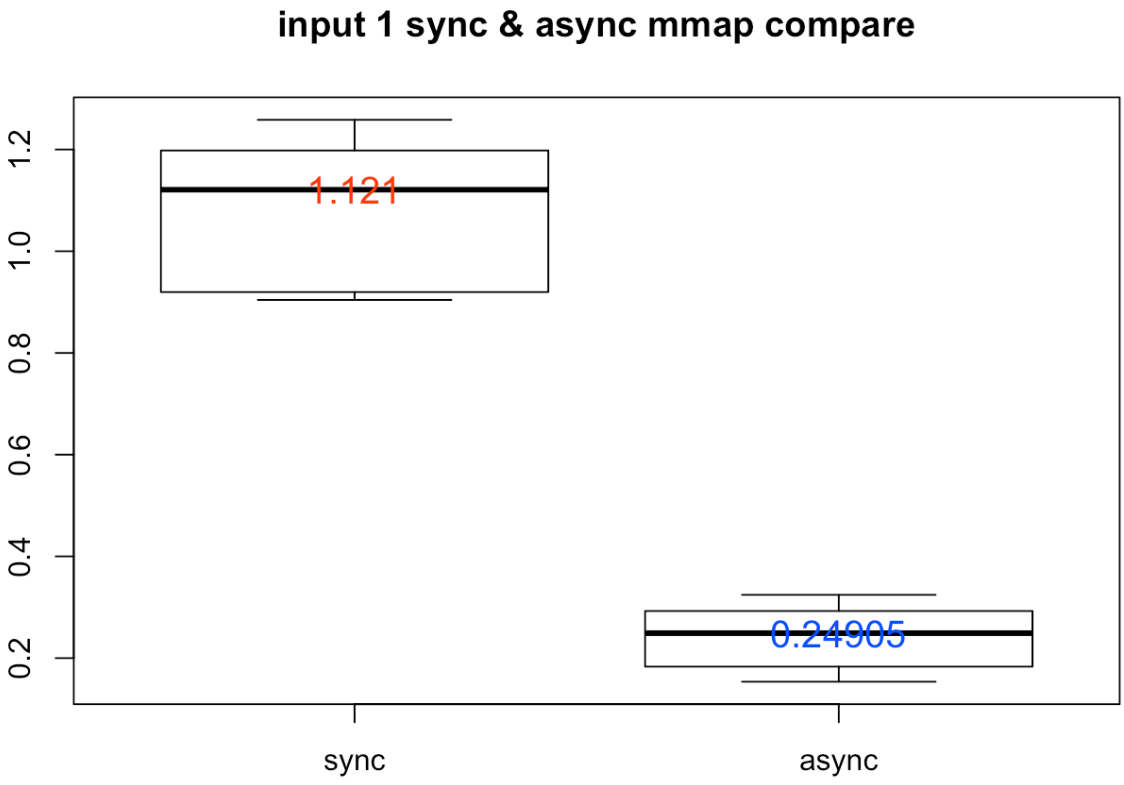
搭配bonus

以下列出不同方法在不同 input 的情形下 synchronous 與 asynchronous 的效能比較

- sample input1 fcntl 的 sync 與 async 的比較

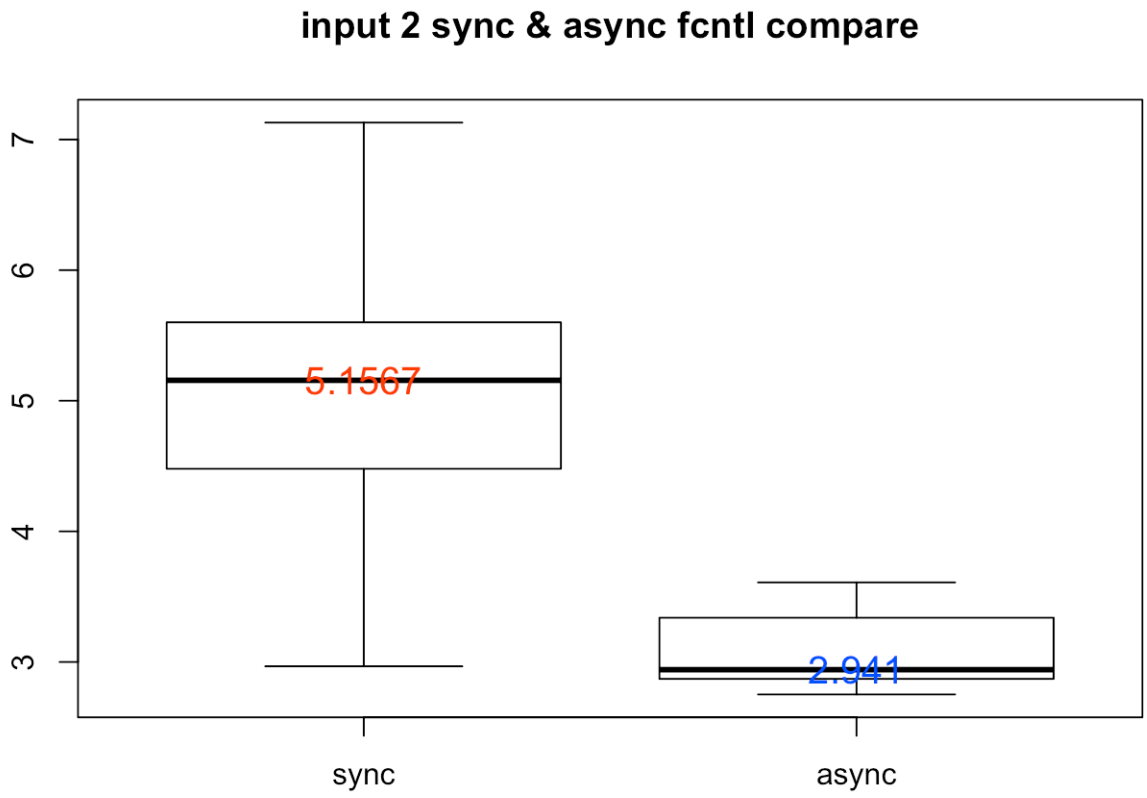


- sample input1 mmap 的 sync 與 async 的比較

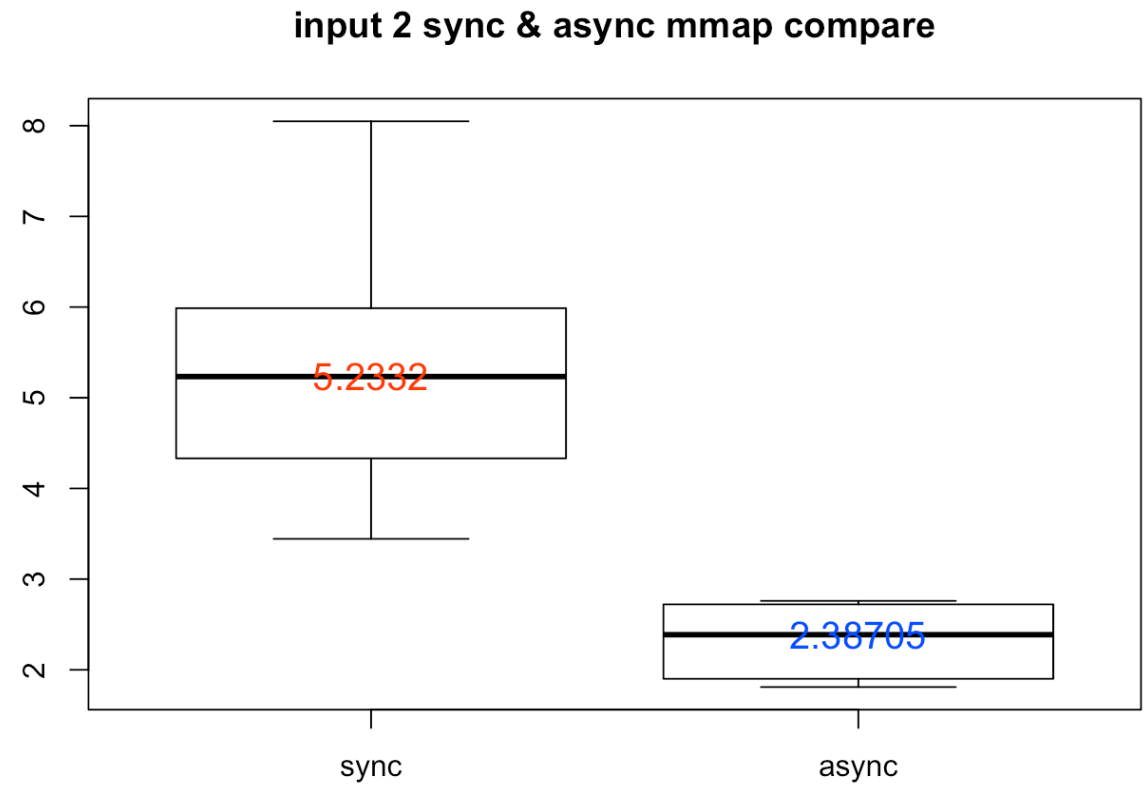


以上兩者是經由多次實驗後畫出的盒方圖，檔案大小為3018 bytes，可以發現在 fcntl-fcntl 及 mmap-mmap 的方法下，若為多個小檔案的傳送，使用 asynchronous 的效果都明顯較 synchronous 好，因為 asynchronous 不需要等所有的 buffer 滿了才清空。

- sample input2 fcntl 的 sync 與 async 的比較

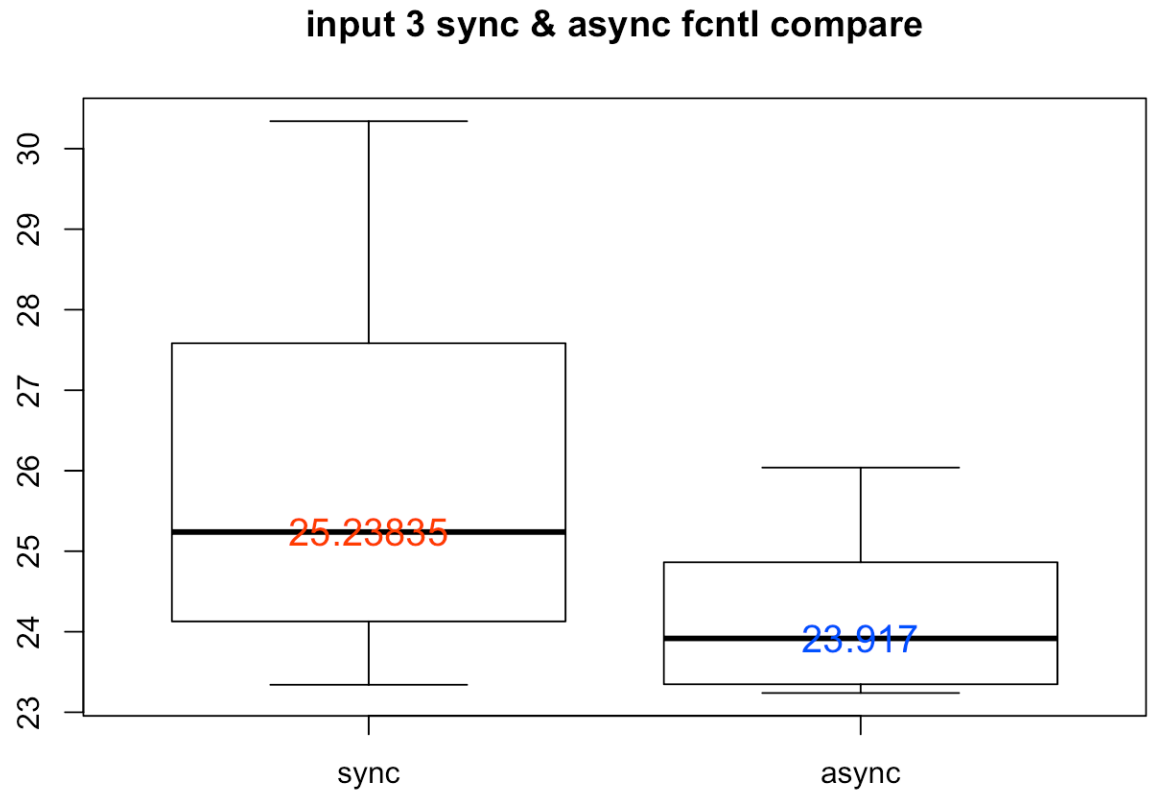


- sample input2 mmap 的 sync 與 async 的比較

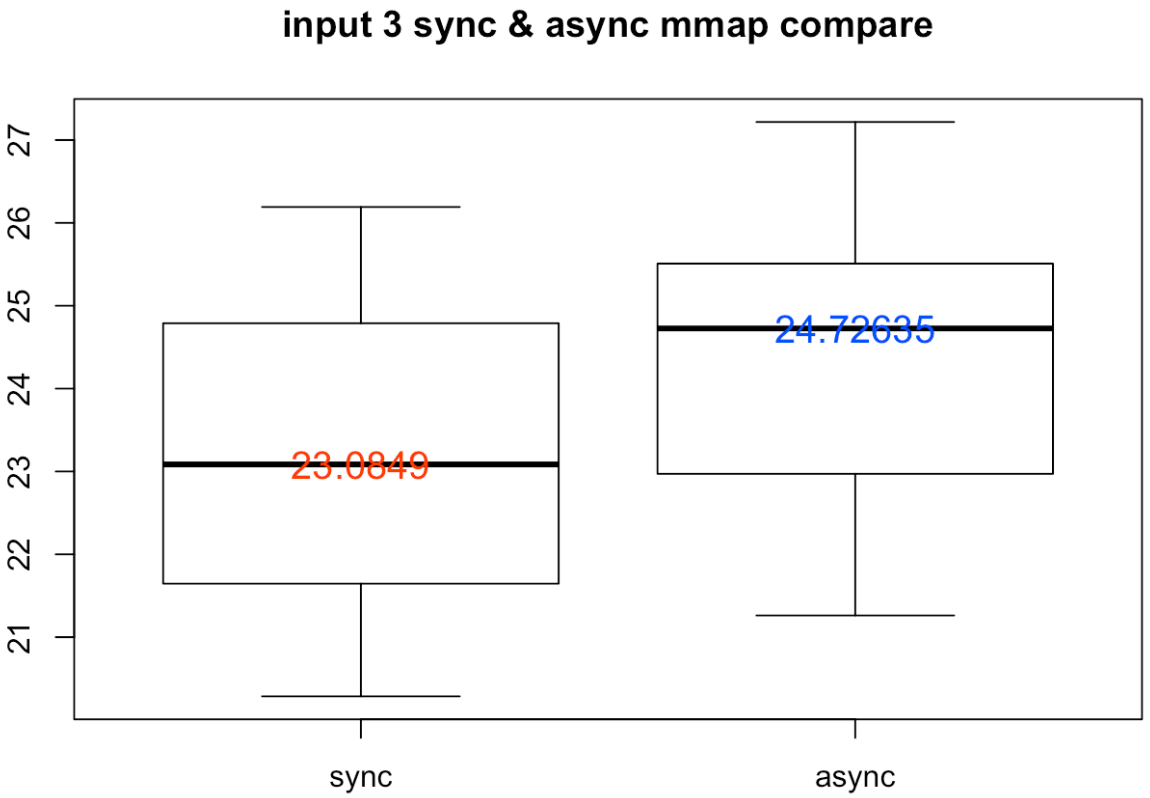


以上兩者是經由多次實驗後畫出的盒方圖，檔案大小為1502860 bytes，可以發現在 fcntl-fcntl 及 mmap-mmap 的方法下，若為一個較大的檔案，使用 asynchronous 的效果都明顯較 synchronous 好，因為 asynchronous 不需要等所有的 buffer 滿了才清空。

- sample input3 fcntl 的 sync 與 async 的比較



- sample input3 mmap 的 sync 與 async 的比較



以上兩者是經由多次實驗後畫出的盒方圖，input 的內容為我們自己產生的測資，總共有5個檔案，總共的檔案大小為9017255 bytes，可以發現在 fcntl-fcntl 的方法下， asynchronous 的效能會較 synchronous 佳，然而 mmap-mmap 的方法下， synchronous 的效能會較佳，會和上方兩個 sample input 的結果不同，推測可能 asynchronous 有加上清空多次 buffer 的 overhead。

- 綜合三種不同大小的 input 可以發現當檔案越大時，mmap-mmap 搭配 asynchronous 的

transmission time 會較 synchronous 慢一些，可能如同上述提到與 buffer size 有關，buffer size 較小會因為清空 buffer 增加 overheaad，而 fcntl-fcntl 搭配 asynchronous 的表現都會較 synchronous 佳。

遇到的問題，及解決方法

問題

- 當檔案太大的時候，用`mmap`時，會 **Segmentation fault (core dumped)**

解決方法

- 每次 `mmap`之後，要`munmap`，否則會佔用太多記憶體，造成記憶體不夠用
  - master.c

```
munmap(file_address, MAP_SIZE);
munmap(kernel_address, MAP_SIZE);
```

  - slave.c

```
munmap(file_add, r);
munmap(kernel_add, r);
```

組內分工表

學號及姓名	分工內容
B07902009 資工二 尚沂瑾	user program slave、bonus 程式實作及report撰寫
B07902031 資工二 黃永雯	user program slave、bonus 程式實作及report撰寫、效能比較分析繪圖及report撰寫
B07902033 資工二 陳郁鳳	kernel program 程式實作及report撰寫
B07902043 資工二 許喬茵	kernel program 程式實作及report撰寫
B07902085 資工二 張琪	user program master 程式實作及report撰寫、效能比較分析
B07902122 資工二 劉彥伶	user program master 程式實作及report撰寫、效能比較分析

Reference

user

mmap

posix_fallocate

munmap

device

module construction

kernel memory mapping

io_remap_pfn_range

vm_area_struct

linux function recv

linux function send