OS Project 2

page descriptors

```
[ 1189.715550] master: 8000000325780267
[ 1189.716022] slave: 8000000325500225
```

master.c’ s mmap section

這段程式碼是傳輸master的input_file到master_device的實作過程：

- 變數
  data_sent : 已經傳輸的bytes
  file_size : 檔案的大小
  src_address : 使用memory mapping 把file_fd mapping到記憶體的file的offset
  dst_address : 使用memory mapping 把dev_fd mapping到記憶體的device的offset
  device的offset
  BUF_SIZE : buffer的大小(512)
  PAGE_SIZE : page的大小(4096)
  MAP_SIZE : 1/10/100/1000 * PAGE_SIZE(一次最大mapping的長度)
- 實作過程
  - 每次使用memcpy從src_address 複製一段長度為len的memory到dst_address（利用ioctl進行I/O控制）。當data_sent達到一個PAGE的大小或檔案已經傳送完畢便跳出去使用munmap釋放原先的src_address，再檢查檔案是否已經全部傳送到device，如果沒有的話便再進行整個流程。

```
64    offset = 0;
65    while (offset < file_size){
66       length = MAP_SIZE;
67       if(file_size - offset < length) length = file_size - offset;
68       if((src = mmap(NULL, length, PROT_READ, MAP_SHARED, file_fd, offset)) == (void *) -1) {
69          perror("master mapping input file");
70          return 1;
71       }
72       if((dst = mmap(NULL, length, PROT_WRITE, MAP_SHARED, dev_fd, offset)) == (void *) -1) {
73          perror("master mapping output device");
74          return 1;
75       }
76       memcpy(dst, src, length);
77       offset += length;
78       ioctl(dev_fd, 0x12345678, length);
79       munmap(src, length);
80       munmap(dst, length);
81    }
```

slave.c mmap section

這段程式碼是傳輸master的input_file到master_device的實作過程：

- 變數:
    - data_sent : 已經傳輸的bytes
    - file_size : 檔案的大小
    - src_address : 使用memory mapping 把file_fd mapping到記憶體的file的offset
    - dst_address : 使用memory mapping 把dev_fd mapping到記憶體的device的offset
    - BUF_SIZE : buffer的大小
    - PAGE_SIZE : page的大小
    - MAP_SIZE : 1/10/100/1000 * PAGE_SIZE(一次最大mapping的長度)
- 實作過程
    - 這段程式碼是在實作從slave_device把資料用mmap讀寫到slave的file上。
    - 使用ioctl進行讀寫控制
    - 使用了mmap和memcpy來實作slave_device讀取和寫入file
    - 使用posix_fallocate這個函式來向系統宣告file所需要的大小
    - 每回合讀寫完畢會使用munmap釋放原本mmap所宣告的記憶體位址

```
59    offset = 0;
60    while(1){
61        length = ioctl(dev_fd, 0x12345678, 0);
62        if(length == 0) break;
63        //printf("slave length: %lu\n", length);
64        if((src = mmap(NULL, length, PROT_READ, MAP_SHARED, dev_fd, offset)) == (void *) -1) {
65            perror("slave mapping input device");
66            return 1;
67        }
68        posix_fallocate(file_fd, offset, length);
69        if((dst = mmap(NULL, length, PROT_WRITE, MAP_SHARED, file_fd, offset)) == (void *) -1) {
70            perror("slave mapping output file");
71            return 1;
72        }
73        memcpy(dst, src, length);
74        ioctl(dev_fd, 0x12345676, (unsigned long)dst);
75        munmap(src, length);
76        munmap(dst, length);
77        offset += length;
78        total_file_size += length;
79    }
```

master_device.c and slave_device.c 共同程式碼

使用自訂的mmap來傳送資料給slave device
重要參數:

- vma: 要mapping的那段虛擬記憶體
- vm_pgoff: 以PAGE_SIZE為單位的offset
- vm_start: vma區塊的start address
- vm_end: vma區塊的end address
- vm_page_prot: 這個vma的權限，如Read, Write, Execute
- vm_flags: vma模式，比如Read, Write, Execute等
- vm_private_data: 指向共享區的指標
- vm_ops: 調用vm函數的指標

```
// for mmap
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) {}

// for mmap
struct vm_operations_struct mmap_vm_ops = {
   .open = mmap_open,
   .close = mmap_close
};

static int my_mmap(struct file *filp, struct vm_area_struct *vma)
{
      vma->vm_pgoff = virt_to_phys(filp->private_data)>>PAGE_SHIFT;
      if (remap_pfn_range(vma, vma->vm_start, vma->vm_pgoff, vma->vm_end - vma->vm_start, vma-
            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的mmap程式碼

- master_open函式主要實作向kernel宣告記憶體
- master_close函式主要實作向kernel釋放已宣告的記憶體
- master_IOCTL_MMAP這裡是向kernel傳送目前剩下的data size

```
int master_close(struct inode *inode, struct file *filp)
{
   kfree(filp->private_data);
   return 0;
}

int master_open(struct inode *inode, struct file *filp)
{
   printk("master is opened!\n");
   filp->private_data = kmalloc(MAP_SIZE, GFP_KERNEL);
   return 0;
}

 case master_IOCTL_MMAP:
   ret = ksend(sockfd_cli, file->private_data, ioctl_param, 0);
   printk("master data_size = %d\n", ret);
   break;
```

slave_device.c的mmap程式碼

- slave_open函式主要實作向kernel宣告記憶體
- slave_close函式主要實作向kernel釋放已宣告的記憶體
- slave_IOCTL_MMAP這裡是向kernel傳送目前接收的的data size
  – 為了避免在master.c使用fcntl而slave.c使用mmap時，會有buffer_size與MAP_SIZE大小不同而造成mmap以為讀到最後一個檔案的錯誤，此處加上一個do-while迴圈，讓每次call ioctl_mmap後都可以拿到整數倍MAP_SIZE大小的檔案

```
int slave_close(struct inode *inode, struct file *filp)
{
   kfree(filp->private_data);
   return 0;
}

int slave_open(struct inode *inode, struct file *filp)
{
   filp->private_data = kmalloc(MAP_SIZE, GFP_KERNEL);
   return 0;
}

case slave_IOCTL_MMAP: // similar to master_device
do {
   rec_piece = krecv(sockfd_cli, file->private_data + rec, MAP_SIZE - rec, 0);
   rec += rec_piece;
} while (rec < MAP_SIZE && rec_piece != 0);

ret = rec;

break;
```

Bonus程式碼

- 只要把socket由sync的模式改成async的模式，修改socket struct的flags即可
- 在ksocket.c 的ksocket和kaccept的函式加入一下程式碼

```
#ifdef BONUS
   sk->flags |= FASYNC;
#endif
```

實作過程中遇見的問題

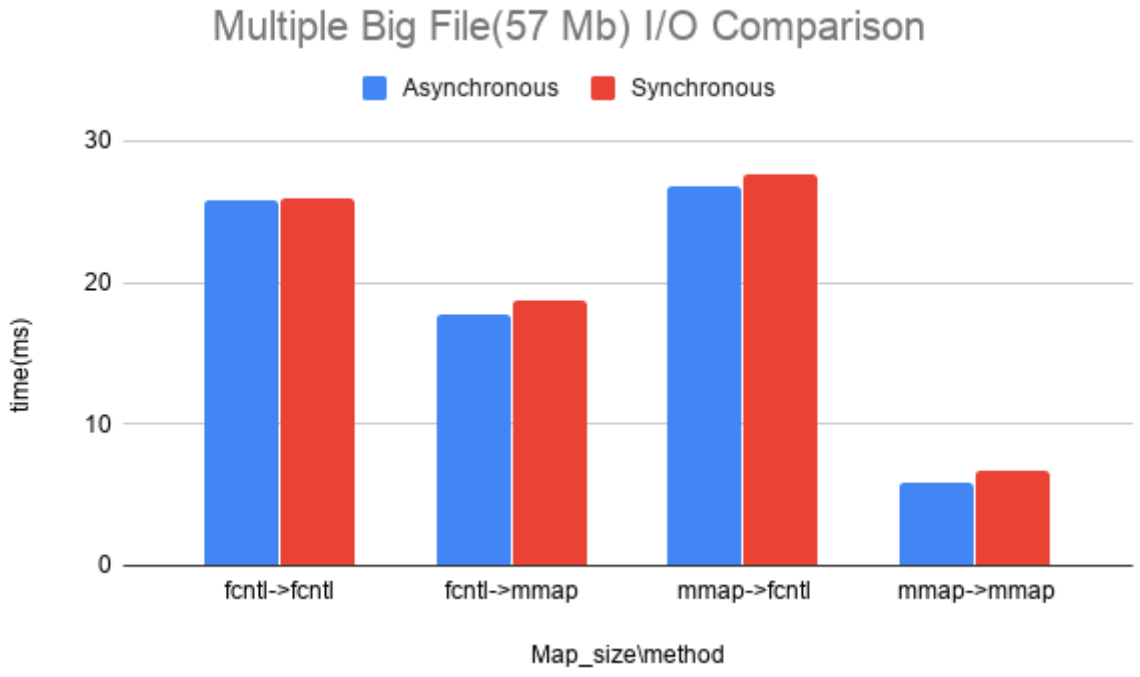
- 在slave.c中，原本使用ftruncate來將output file需要的空間allocate出來以供mmap使用，但會出現問題，改成posix_fallocate後可以解決

- 在master.c使用fcntl而slave.c使用mmap時，會有buffer_size與MAP_SIZE大小不同而造成mmap以為讀到最後一個檔案的錯誤。加上一個do-while迴圈於slave_device.c，讓每次call ioctl_mmap後都可以拿到整數倍MAP_SIZE大小的檔案後便可以解決

Statictical Analysis

Synchronous				
Map_size\method	fcntl->fcntl	fcntl->mmap	mmap->fcntl	mmap->mmap
1 small file (32 bytes)	0.044	0.05	0.052	0.038
multiple small file (933 bytes)	0.17	0.1	0.21	0.125
1 big file (12 Mbytes)	5.34	3.34	5.26	1.2
multiple big file (57 Mbytes)	26	18.72	27.67	6.73
Asynchronous				
Map_size\method	fcntl->fcntl	fcntl->mmap	mmap->fcntl	mmap->mmap
1 small file (32 bytes)	0.042	0.03	0.041	0.051
multiple small file (933 bytes)	0.16	0.157	0.203	0.183
1 big file (12 Mbytes)	5.62	3.6	5.13	1.16
multiple big file (57 Mbytes)	25.77	17.7	26.8	5.9

- 由上圖可以看出Memory Mapping在小檔案時和fcntl的速度不相上下，但在傳大檔案的時候Memory Mapping的速度就有明顯的變快，在一次傳多個大檔案時Memory Mapping的速度優勢就變得非常明顯了

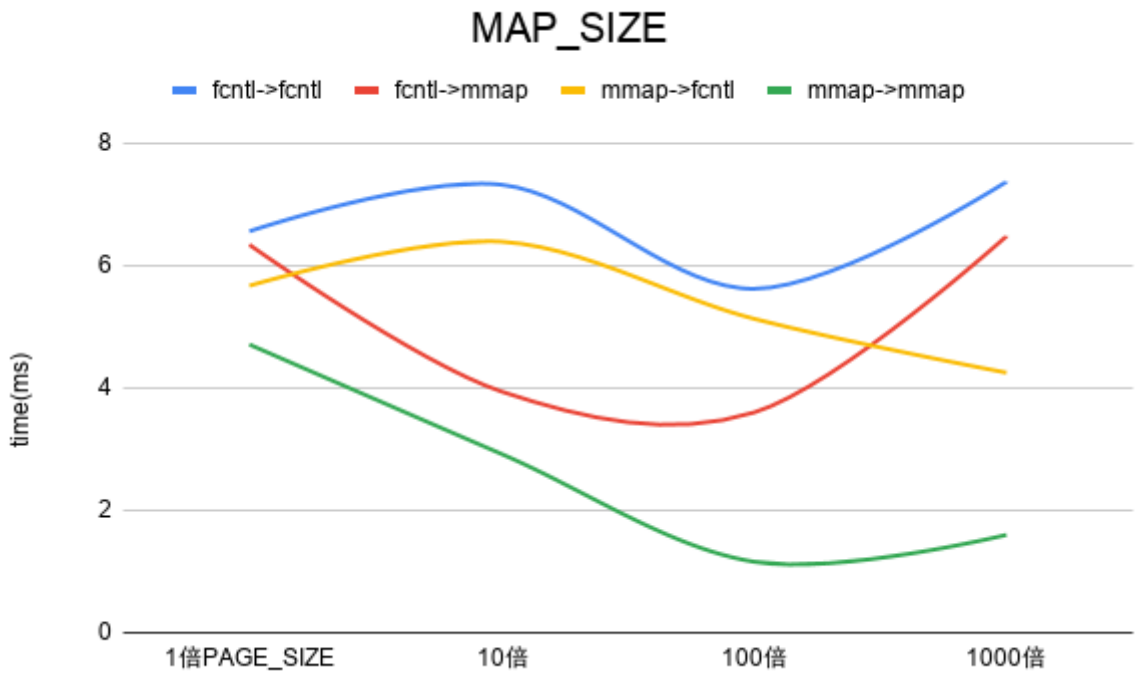


- 使用Asynchronous方法和Synchronous相比快了一些，在使用fcntl來傳資料時Asynchronous變快的不多，但使用Memory Mapping時Asynchronous增加的速度就比較明顯了。

MAP_SIZE對時間的影響(檔案大小: 12Mb)

統計不同大小的Mapsize以後可以得出100倍左右的PAGE_SIZE可以有最快的效率。

Map_size\method	fcntl->fcntl	fcntl->mmap	mmap->fcntl	mmap->mmap
1倍PAGE_SIZE	6.56	6.34	5.67	4.71
10倍	7.32	3.94	6.39	2.92
100倍	5.62	3.6	5.13	1.16
1000倍	7.37	6.48	4.25	1.6



組內分工表

學號	姓名	負責項目	分工比重
b07902124	鄭世朋	coding, debug	10
b07902014	蔡承濬	coding, debug	10
b07902011	許耀文	coding, debug	10
b07902109	張翔文	report, coding, debug	10
b07902089	李智源	report, coding, Bonus	10
b07902048	李宥霆	report, coding, debug	10

Reference

- b07902022 張鈞堯
- https://github.com/wangyenjen/OS-Project-2 (https://github.com/wangyenjen/OS-Project-2)

- https://github.com/andy920262/OS2016/tree/master/project2 (https://github.com/andy920262/OS2016/tree/master/project2)