OS Project 2 A/Synchronous Virtual Device

- b07902055 - 謝宗晅
- b07902053 - 許浩鳴
- b07902041 - 邱榆洋
- b07902027 - 高偉倫
- b07902103 - 李建霖
- b07902029 - 王泳錡

1. Programing design

Device 修改

我們在 master device 和 slave device 中，增加了 memory-map 的部分：

```
static struct file_operations master_fops = {
    .owner = THIS_MODULE,
    .unlocked_ioctl = master_ioctl, // using custom ioctl
    .open = master_open, // method for opening the device
    .write = send_f, // method for writing to the device
    .release = master_close, // method when closing the device
    .mmap = memory_mapping // method for mapping device memory
};

struct vm_operations_struct mmap_vm_ops = {
    .open = mmap_open,
    .close = mmap_close,
    .fault = mmap_fault
};

void mmap_open(struct vm_area_struct *vma) {}
void mmap_close(struct vm_area_struct *vma) {}

static int memory_mapping(struct file *filp, struct vm_area_struct *vma) {
    // vma->vm_start: starting virtual address
    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->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;
}
```

此外為了避免 版本相容性問題，mmap的page_fault 分別處理：
(雖然理論上不會出現 page fault)

```
#if LINUX_VERSION_CODE < KERNEL_VERSION(4,11,0)
static int mmap_fault(struct vm_area_struct *vma, struct vm_fault *vmf) {
    vmf->page = virt_to_page(vma->vm_private_data);
    get_page(vmf->page);
    return 0;
}
#elif LINUX_VERSION_CODE >= KERNEL_VERSION(5,3,0)
static vm_fault_t mmap_fault(struct vm_fault *vmf) {
    vmf->page = virt_to_page(vmf->vma->vm_private_data);
    get_page(vmf->page);
    return 0;
}
#else
static int mmap_fault(struct vm_fault *vmf) {
    vmf->page = virt_to_page(vmf->vma->vm_private_data);
    get_page(vmf->page);
    return 0;
}
#endif
```

而至於兩個device分別各自處理 mmap 的部份：

```
case MASTER_MMAP:
    ret = ksend(sockfd_cli, filp->private_data, arg, 0);
    break;
```

```
case SLAVE_MMAP:
    ret = krecv(sockfd_cli, filp->private_data, arg, 0);
    break;
```

其中， `arg` 是我們指定 device 要 map 的長度。

並且因為 `sample_code` 裡面的 `slave_device` 本來的寫法是一次直接讀 `BUF_SIZE` 的大小，但是因為我們需要控制一次讀的大小，所以就將寫法改為與 `count` 有關 (雖然 `sample_code` 的寫法在某些情況下可以 work，但是無視 `read()` 所傳入的 `count` 引數實在不是一個正常的寫法)：

```
ssize_t receive_msg(struct file *filp, char *buf, size_t count, loff_t *offp) {
    // call when user is reading from this device
    char msg[BUF_SIZE];
    size_t len;
    /*
     * change from "sizeof(msg)" (which equals to BUF_SIZE)
     * to "count * sizeof(char)"
     */
    len = krecv(sockfd_cli, msg, count * sizeof(char), 0);
    if (copy_to_user(buf, msg, len))
        return -ENOMEM;
    return len;
}
```

然而 `master_device` 不用做這樣的修改，因為 `master_device` 中的寫法本來就是有使用 `count` 引數的寫法。

Master/Slave之間的多檔傳輸

- 如果沒有定義好檔案界線，很容易出現race condition:
  - master 一次將 file1/file2/file3 不間斷地傳進 master device
  - slave 一直從 slave device 讀檔，導致讀 file1 時讀到 file1/file2/file3 的內容
- 為了防止以上狀況，我們有兩種做法：

1. master 端必須要等待 slave 端讀完 device 中的所有內容，才能繼續傳下一個檔。
2. master 和 slave 之間溝通好：master 在每次傳檔的時候先傳檔案大小，slave 依據此大小讀檔。

- 我們採用的是方法二，也就是：
  - master端：在傳檔案本體前先傳一個 `off_t` 大小的變數進去，代表檔案長度。
  - slave端：先讀進一個 `off_t` 大小的變數代表檔案長度，再依據此變數的值讀進對應長度的檔案。

master.c

將原本 sample code 改成多檔處理，並將 method 存好

```
for (int i = 2; i < argc - 1; i++) {
     strcpy(file_name[i - 2], argv[i]);
     fprintf(stderr, "file name : %s\n", file_name[i - 2]);
}

file_num = atoi(argv[1]);

strcpy(method, argv[argc-1]);
```

之後在 method 中增加 mmap 的方法，並且使用 for 迴圈處理多個檔案

```
for (int i = 0; i < file_num; i++) {
    switch(method[0]) {
        case 'm':{ // mmap
            // transfer file_size
            off_t offset = 0;
            kernel_address = mmap(NULL, MMAP_SIZE, PROT_READ | PROT_WRITE,
                                  MAP_SHARED, dev_fd, offset);
            memcpy(kernel_address, &file_size, sizeof(off_t));
            while(ioctl(dev_fd, 0x12345678, sizeof(off_t)) < 0
                  && errno == EAGAIN);

            while (offset < true_file_size) {
                off_t len = MIN(MMAP_SIZE, file_size);
                file_address = (char *)mmap(NULL, MMAP_SIZE, PROT_READ,
                                            MAP_SHARED, file_fd, offset);
                kernel_address = (char *)mmap(NULL, MMAP_SIZE,
                                              PROT_READ | PROT_WRITE,
                                              MAP_SHARED, dev_fd, offset);
                memcpy(kernel_address, file_address, len);
                while (ioctl(dev_fd, 0x12345678, len) < 0 && errno == EAGAIN);
                offset += len;
                file_size -= len;
            }
            if (ioctl(dev_fd, 0x12345676, kernel_address) == -1)
                perror("ioctl server");
            // munmap
            munmap(kernel_address, PAGE_SIZE);
            break;
        }
    }
}
```

Master 要做的事情如下：

- 將檔案大小 `mmap()` 並且 `memcpy()` 到 memory
- `mmap()` 之後要 call `ioctl()` 來讓 master device 傳東西到 slave device
- 再每次都把 file 的一部分都 `mmap()` 並且 `memcpy()` 到 memory，並且 call `ioctl()` 來啟動傳輸

slave.c

同 master ，先處理多檔部分

```
  for (int i = 2; i < argc - 1; i++) {
      strcpy(file_name[i - 2], argv[i]);
      fprintf(stderr, "file name : %s\n", file_name[i-2]);
  }

  file_num = atoi(argv[1]);

  strcpy(method, argv[argc-1]);
```

而 method 的部分，除了增加了 mmap ，也稍微修改了 fcntl 的部分

```
for (int file_index = 0; file_index < file_num; file_index++) {
    if ((file_fd = open(file_name[file_index],
         O_RDWR | O_CREAT | O_TRUNC)) < 0) {
        perror("failed to open input file\n");
        return 1;
    }

    // get start time
    gettimeofday(&start, NULL);

    // start transfering
    switch (method[0]) {
        case 'f': { // fcntl : read()/write()
            // read file size
            ret = read(dev_fd, &file_size, sizeof(size_t));

            true_file_size = file_size;
            // open file
            while (file_size > BUF_SIZE) {
                ret = read(dev_fd, buf, sizeof(buf));
                write(file_fd, buf, ret); // write to the input file
                file_size -= ret;
            }
            // there're only 'file_size' characters left to be read
            ret = read(dev_fd, buf, file_size * sizeof(char));
            write(file_fd, buf, ret);
            break;
        }
        case 'm': { // mmap
            off_t offset = 0;

            // get file_size
            while ((ioctl(dev_fd, 0x12345678, sizeof(off_t))) < 0
                    && errno == EAGAIN);
            kernel_address = mmap(NULL, MMAP_SIZE, PROT_READ,
                                  MAP_SHARED, dev_fd, offset);
            memcpy(&file_size, kernel_address, sizeof(off_t));
            true_file_size = file_size;

            while (offset < true_file_size) {
                off_t len = MIN(MMAP_SIZE, file_size);
                posix_fallocate(file_fd, offset, len);
                printf("len = %ld\n", len);
                while((ioctl(dev_fd, 0x12345678, len)) < 0 && errno == EAGAIN);
                kernel_address = (char *)mmap(NULL, MMAP_SIZE, PROT_READ,
                                              MAP_SHARED, dev_fd, offset);
                file_address = (char *)mmap(NULL, MMAP_SIZE,
                                            PROT_WRITE | PROT_READ,
                                            MAP_SHARED, file_fd, offset);
                memcpy(file_address, kernel_address, len);
                munmap(file_address, len);
                offset += len;
                file_size -= len;
            }
            ftruncate(file_fd, true_file_size);
            munmap(kernel_address, PAGE_SIZE);
            break;
        }
        close(file_fd);
    }
}
```

Slave 要做的事情如下：

- 先 call `ioctl()` 來從 `dev_fd` 接收資料
- 再 call `mmap()` 並且 `memcpy()` 來獲得檔案大小

- 之後每次都先 call `ioctl()` 來接收 file 的一部分內容，再 `mmap()` 並且 `memcpy()` 到 output file

Master 和 Slave 都使用了一樣的方法來傳輸/接收資料，在 `while()` 迴圈裡，每一次都把 file size 減掉已經傳輸的 byte 數，如果已經全部傳輸完畢的話就直接離開 `while()` 迴圈。

mmap's page descriptor

我們可以看到實體記憶體的位置

Bonus-Asynchronous Socket Programming

另外，在 bonus 的部分，我們在 ksocket.c 中增加了 async 的部分

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

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

        return sk;
}
```

我們知道，device 在 kernel space 裡執行，而 program 則是在 user space 裡執行，兩者無法互相溝通。但實際上，在設備準備好的情況下，我們希望通知 user 可以 read 了，這樣 program 就不需要一直去 trap in system 查看當前 device 的狀況，進而節省 resource ，這就是 asynchronous 。fasync 就是為了使 device 讀寫和 program 的讀寫分開，使得 program 可以在 device 讀寫的時候做別的事。

增加 FASYNC 這個 flag 將呼叫以下 system call 來實現 fasync 方法：

```
int fasync_helper(struct inode *inode, struct file *filp,
                  int mode, struct fasync_struct **fa);
```

2. Compare File I/O to memory-mapped I/O

比較兩者結果與效能

結果數據呈現

fcntl - sync (different sizes)

	4B	16B	64B	256B	4KB	16KB	64KB	256KB
Master	0.024	0.043	0.022	0.021	1.05	0.077	2.916	24.603
Slave	0.013	0.012	0.013	0.013	0.04	0.104	3.197	30.361

	4MB	16MB	64MB	256MB
Master	36.737	130.862	606.385	3385.460
Slave	41.941	160.648	635.939	3440.915

mmap - sync (different sizes)

	4B	16B	64B	256B	4KB	16KB	64KB	256KB
Master	0.043	0.034	0.054	0.043	0.048	0.070	0.138	0.422
Slave	0.116	0.244	0.185	0.251	0.327	0.175	0.416	1.881

	4MB	16MB	64MB	256MB
Master	23.670	87.113	439.006	1758.392
Slave	44.324	129.228	483.082	1813.969

fcntl - different buffer

data(KB)/buf(B)	16/64	16/256	16/512	16/1024
Master	0.080	0.135	0.082	0.070
Slave	0.272	0.177	0.107	0.068

data(KB)/buf(B)	64/64	64/256	64/512	64/1024
Master	2.904	1.530	0.258	0.149
Slave	3.141	0.674	0.506	0.199

data(KB)/buf(B)	256/64	256/256	256/512	256/1024
Master	1.374	1.879	0.940	0.526
Slave	6.578	5.753	1.387	0.770

data(KB)/buf(B)	1024/64	1024/256	1024/512	1024/1024
Master	71.700	19.526	16.759	16.917
Slave	85.607	20.636	19.820	18.908

fcntl - async

	4B	16B	64B	256B	4KB	16KB	64KB	256KB
Master	0.022	0.034	0.025	0.024	0.038	0.079	0.263	10.161
Slave	0.012	0.015	0.012	0.014	0.040	0.103	0.356	9.853

	4MB	16MB	64MB	256MB
Master	34.806	127.706	611.302	2847.542
Slave	36.682	160.087	641.954	2887.974

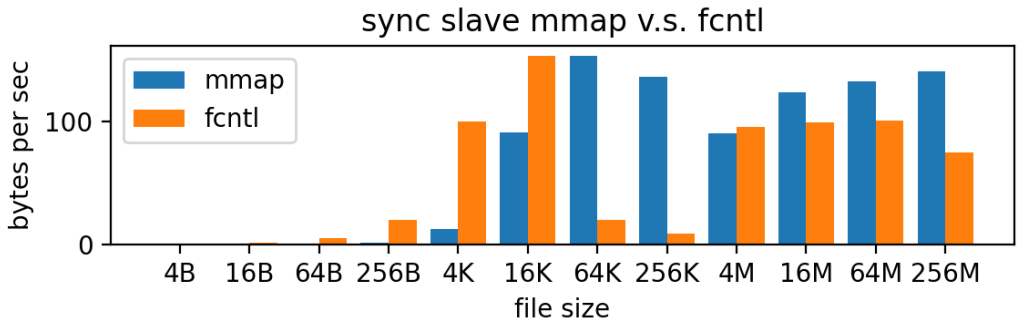
mmap - async

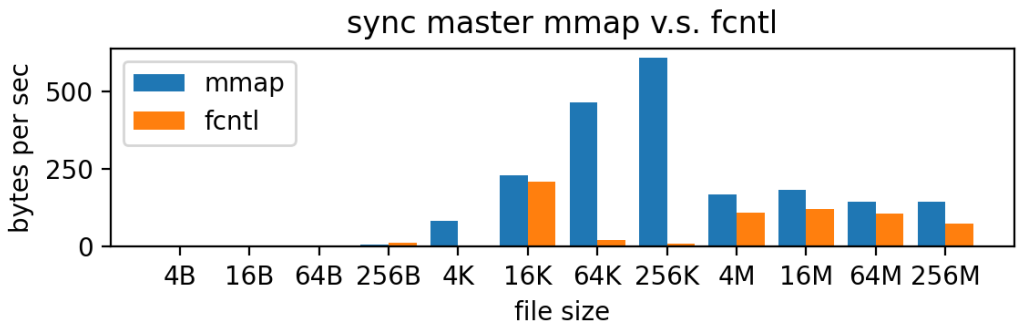
	4B	16B	64B	256B	4KB	16KB	64KB	256KB
Master	0.036	0.039	0.036	0.034	0.038	0.054	0.134	0.408
Slave	0.122	0.115	0.132	0.119	0.125	0.180	0.355	1.175

	4MB	16MB	64MB	256MB
Master	21.544	92.3020	416.170	2360.823
Slave	39.623	125.311	458.393	2437.105

比較：

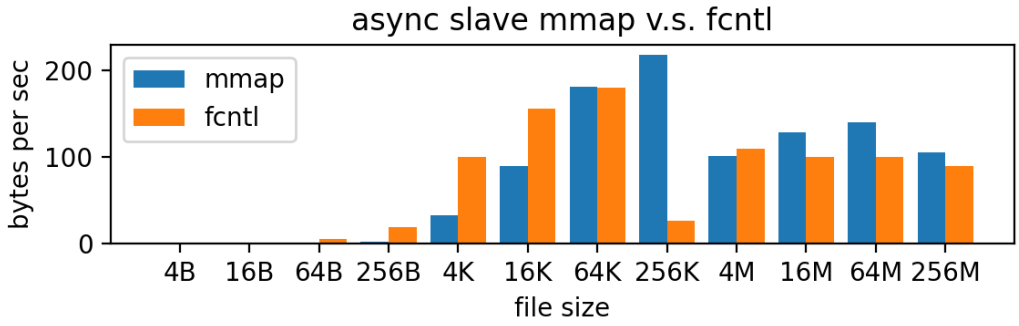
(sync) fcntl vs mmap

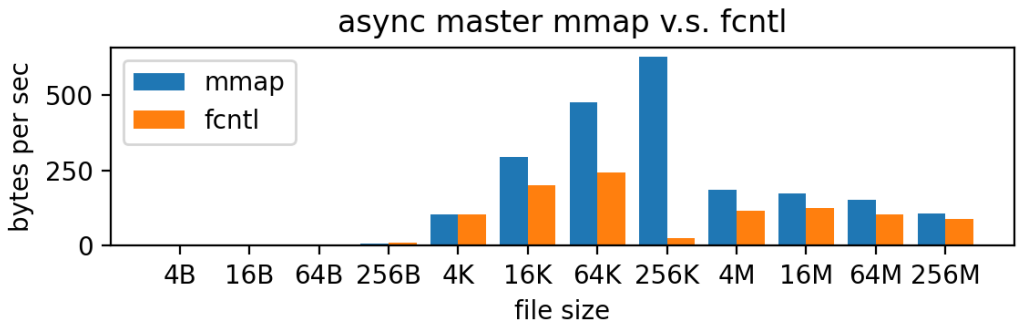




由以上可知，mmap 檔案大的時候比較快，只是在檔案小的時候可能會有 overhead 的問題，反而稍慢於fcntl

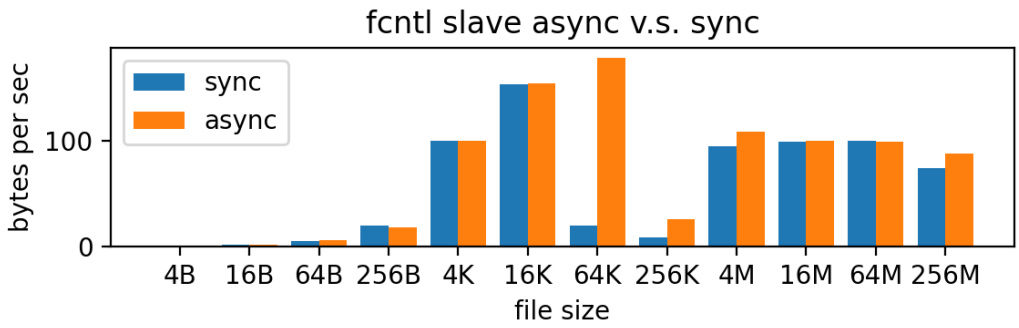
(async) fcntl vs mmap

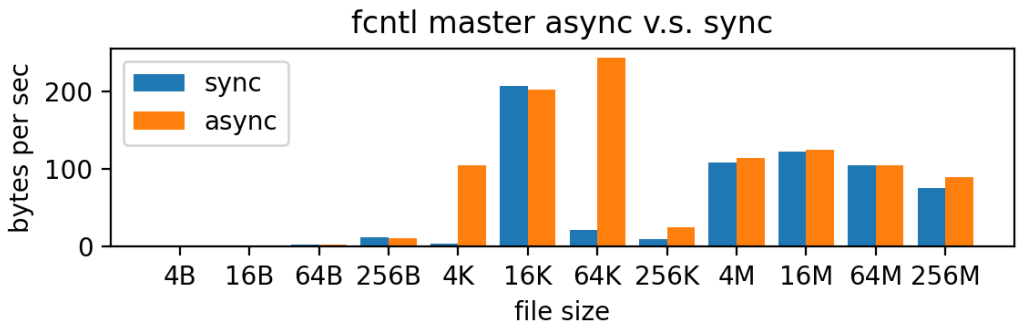




明顯地， mmap 會快一些些，與上述概念類似，都會有 overhead 的問題

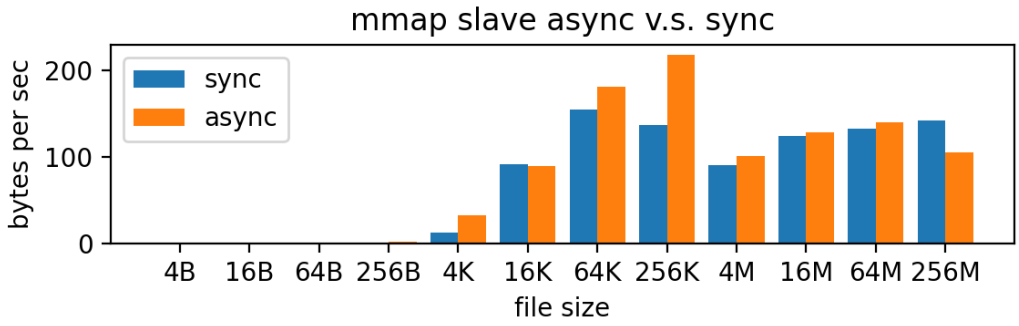
(fcntl) sync vs async

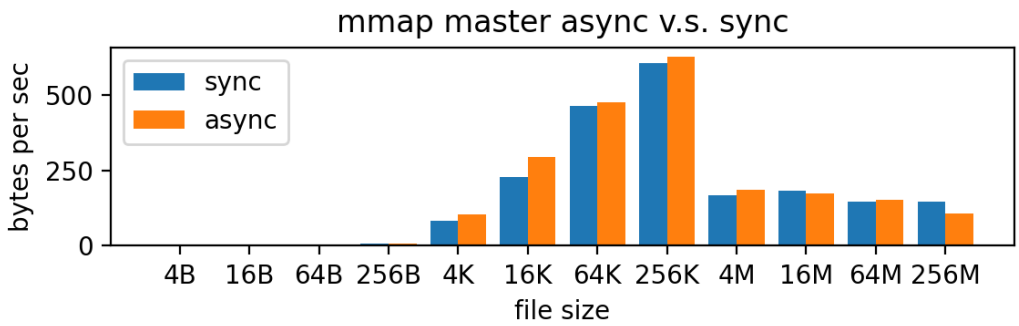




明顯地， async 會快一些些

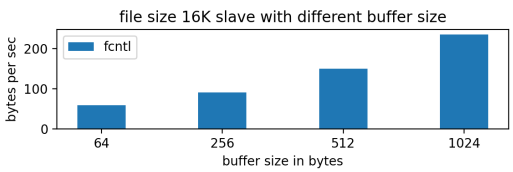
(mmap) sync vs async

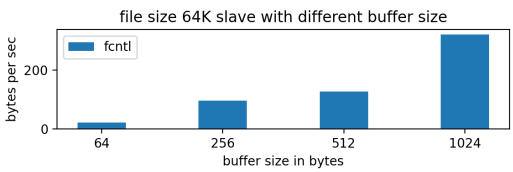


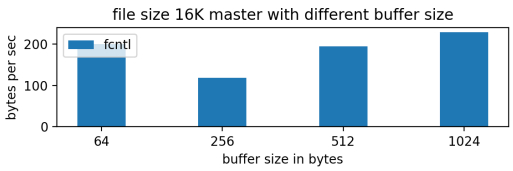


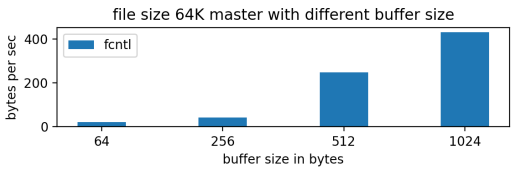
明顯地， async 會快一些些

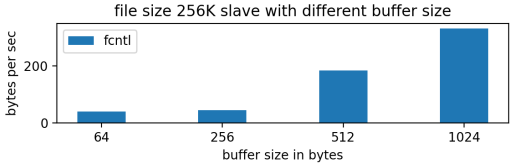
different sizes of the buffer

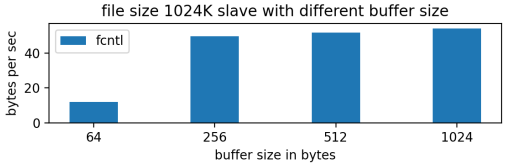


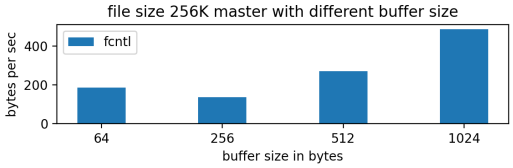


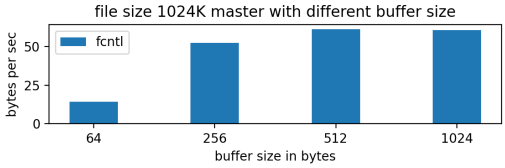












很明顯，如果buffer 越大 ，傳輸的速度越快

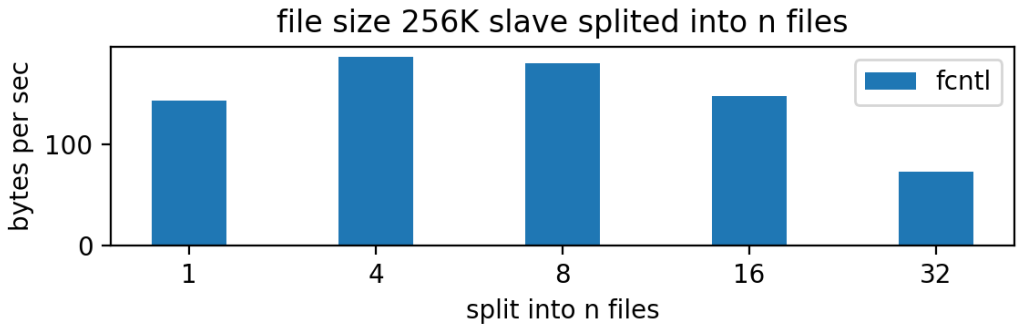
3. Special Idea

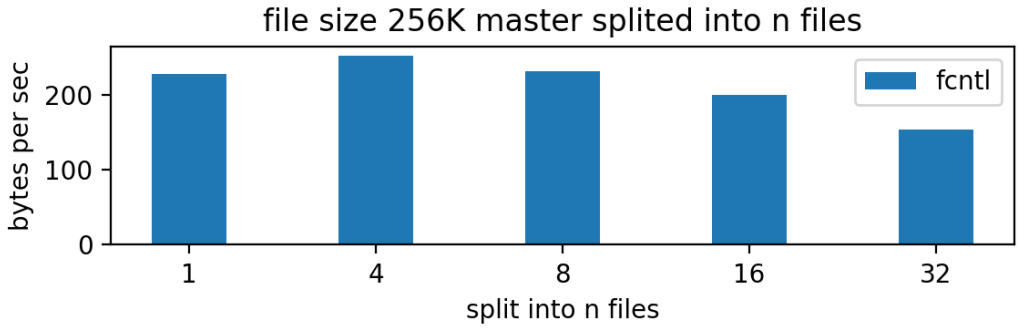
fcntl - split

我們將檔案切開來傳傳看去比較速度

	256KB/1	256KB/4	256KB/8	256KB/16	256KB/32
Master	1.122	0.974	1.004	2.767	3.243
Slave	1.780	2.854	0.185	2.371	3.097

得到統計圖如下：





由上圖發現，256KB的檔案在切成4份的時候特別快，這可能會是一個優化的好方向，
如果每個檔案都去試著找符合自己大小的切分（參數），可能就可以讓這個實驗達到最佳化！

4. 分工

- 謝宗晅：debug, 部分 report, 部分 `user_program/master.c` , 部分 `user_program/slave.c` , `bonus.sh`
- 李建霖: `user_program/master.c` , `user_program/slave.c` , demo
- 許浩鳴: `kernel_device/master.c` , `kernel_device/slave.c` , asynchronous ksocket, `compile.sh`, `clean.sh` , 部分 report
- 高偉倫: 部分 `user_program/slave.c` , 部分report, 產生實驗數據/script (`eval.sh` , `fcntl/mmap_run.sh`)
- 王泳錡: `data_generator.c` , 部分 `report` , 實驗數據表格整理
- 邱榆洋: `kernel_device/master.c` , `kernel_device/slave.c` , bug處理

5. 參考資料

- 助教的 sample code
- How to pass parameters from makefile to linux kernel module source code (https://stackoverflow.com/questions/15430921/how-to-pass-parameters-from-makefile-to-linux-kernel-module-

source-code?fbclid=IwAR1u1YE0jpUPqcLErXk9oSMsfpe7TikscyvMSf7ea3n1eQDoNgELm8v4VJY)