OS Project 2 Report

Design

slave_device.py

The slave device will recieve data from master device through socket.

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

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

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

master_device.py

The master device will allocate some kernel memory to store data and send them to slave device through socket.

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

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

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

Both slave_device.py and master_device.c

We design our `mmap()` function by `remap_pfn_range()` and assign values to vma(virtual memory area) attributes.

```
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->vm_page_prot))
        return -EAGAIN;
    vma->vm_flags |= VM_RESERVED;
    vma->vm_private_data = filp->private_data;
    vma->vm_ops = &mmap_vm_ops;
    mmap_open(vma);
    return 0;
}
```

master.c

The master-side program will read the input file with file descriptor `file_fd` by the specified method `mmap` and send the data to the master device.

```
case 'm':
    dst = mmap(NULL, MAP_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, dev_fd, 0);
    for (int j = 0; j * MAP_SIZE < file_size; j++)
    {
        // map memory of size MAP_SIZE every loop
        // if remaining file size is less than MAP_SIZE, then let mmap_size be the remaining file size
        mmap_size = (file_size - j * MAP_SIZE > MAP_SIZE)? MAP_SIZE : file_size - j * MAP_SIZE;
        src = mmap(NULL, mmap_size, PROT_READ, MAP_SHARED, file_fd, j * MAP_SIZE);
        memcpy(dst, src, mmap_size);
        munmap(src, mmap_size);
```

```c
        while (ioctl(dev_fd, 0x12345678, mmap_size) < 0 && errno == EAGAIN);
        }

    if (ioctl(dev_fd, 0, dst) == -1)
    {
        perror("ioclt server error\n");
        return 1;
    }

    munmap(dst, MAP_SIZE);
    break;
```

slave.c

The slave-side program receives the data with size `MAP_SIZE` each loop from the slave device and writes all of them to the output file by the specified method `mmap()`. We also use `posix_fallocate()` to ensure that disk space is allocated for the file referred to by the file descriptor `fd` for the bytes in the range starting at `fileSize` and continuing for `ret` bytes.

```c
case 'm':
    src = mmap(NULL, MAP_SIZE, PROT_READ, MAP_SHARED, dev_fd, 0);
    while (1)
    {
        while ((ret = ioctl(dev_fd, 0x12345678)) < 0 && errno == EAGAIN);

        if (ret < 0)
        {
            perror("ioctl error\n");
            return 1;
        }
        else if (ret == 0)
            break;
        else
        {
            offset = (size_t)(file_size / PAGE_SIZE) * PAGE_SIZE;

            posix_fallocate(file_fd, file_size, ret);
            dst = mmap(NULL, ret, PROT_WRITE, MAP_SHARED, file_fd, offset);
            memcpy(dst, src, ret);
            munmap(dst, ret);

            file_size += ret;
        }
    }
    ftruncate(file_fd, file_size);

    if (ioctl(dev_fd, 0, src) == -1)
    {
        perror("ioctl error\n");
```

```
            return 1;
        }

        munmap(src, MAP_SIZE);
        break;
```

Bonus

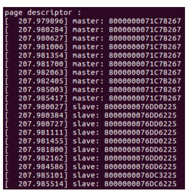
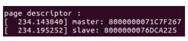
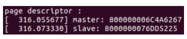
We modify `kernel.c` which adapts to async IO. We add `FASYNC` to the flags when initializing each socket struct in `ksocket.c` to implement async socket.

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

Comparison

As the experiments showed below, we can see that transmit file through mmap method is faster than I/O method usually because *mmap* only modify the virtual memory mapping but *fcntl* access the disk. However, fcntl is slightly faster than mmap in the `.iso` case. We guess that different file extensions may cause different ways of allocating/accessing memory so that it is inconsistent with our expected results.

Using async IO, the transmission efficiency has improved in some files, but files like .mp4 and .iso barely improved. We think that async IO may empty the buffer more times than sync IO, which may cause some overhead.

Transmit file	**target_file_1 (10 files)**	**target_file_2**	**our file (.iso)**	**our file (.pdf)**	**our file (.mp4)**
sync *fcntl* IO transmit time	2.531 ms	12.790 ms	19021 ms	25.581 ms	2038 ms
sync *mmap* IO transmit time	0.586 ms 	8.290 ms 	19033 ms 	11.524 ms 	1943 ms 
async *fcntl* IO transmit time	1.812 ms	9.751 ms	19039 ms	13.213 ms	2003 ms

async *mmap* IO transmit time	0.345 ms	6.344 ms	18961 ms	9.425 ms	1954 ms

Member Contribution

B05902010	張頌平	25%
B05902023	李澤諺	25%
B05902040	宋易軒	25%
B05902116	陳昱鈞	25%

Reference

1. mmap()
2. kmalloc() / kfree()
3. posix_fallocate()