Block Device Driver

Block device driver

Driver 종류	설 명	등록함수명
Char Driver	**device**를 **file**처럼 취급하고 접근하여 직접 **read/write**를 수행, **data** 형태는 **stream** 방식으로 전송 **EX) console, keyboard, serial port driver**등	**register_ chrdev()**
Block Driver	**disk**와 같이 **file system**을 기반으로 일정한 **block** 단위로 **data read/write**를 수행 **EX) floppy disk, hard disk, CD-ROM driver** 등	**register_ blkdev()**
Network Driver	**network**의 **physical layer**와 **frame** 단위의 데이터를 송수신 **EX) Ethernet device driver(eth0)**	**register_ netdev()**

RamDisk

- RamDisk
  - ✓메모리의 일부분을 디스크처럼 사용하는 것
  - ✓파일을 읽고 쓰는 저장장치처럼 사용하는 것
  - ✓디스크의 빠른 IO 처리를 위해 사용

Block Device Driver

헤더파일

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

#include <linux/fs.h>            //register_blkdev()
#include <linux/types.h>         //GFP_KERNEL
#include <linux/fcntl.h>
#include <linux/vmalloc.h>       //vmalloc()
#include <linux/hdreg.h>
#include <linux/blkdev.h>        //block_device_operations, gendisk, request_queue, bio,
#include <linux/blkpg.h>
#include <linux/uaccess.h>
```

전처리

```
#define VRD_DEV_NAME     "vrd"    //드라이버의 이름
#define VRD_DEV_MAJOR    240      //드라이버의 메이저번호

#define VRD_MAX_DEVICE   1        //드라이버를 사용할 디바이스의 개수

#define VRD_SECTOR_SIZE 512                          //디스크의 하나섹터 크기
#define VRD_SIZE        (4*1024*1024)                //디스크의 총 크기
#define VRD_SECTOR_TOTAL (VRD_SIZE/VRD_SECTOR_SIZE) //디스크의 총 섹터의 수
```

Block Device Driver

init()

```
int vrd_init(void){
        vdisk = vmalloc(VRD_SIZE);//램디스크이기 때문에 메모리에서 해당 디바이스의 전체용량을 할당.

        int state = register_blkdev(VRD_DEV_MAJOR, VRD_DEV_NAME); //블록 디바이스 드라이버를 커널에 빌드.
        if(state < 0){
                printk(KERN_INFO "driver build failed..\n");
                return -1;
        }

        device.data = vdisk;//할당한 메모리주소를 make_request()함수에서 사용할 수 있도록 device.data에 설정.
        device.gd = alloc_disk(1);//alloc_disk()함수로 블록 디바이스를 등록하기 위한 정보 구조체인 gendisk구조체를 할당받는다.
        device.queue = blk_alloc_queue(GFP_KERNEL);//vrd_make_request에서 사용하는 블록 디바이스에 필요한 요구 큐를blk_alloc_queue함수로 할당
        blk_queue_make_request(device.queue, &vrd_make_request);//blk_queue_make_request()함수를 이용해 커널과 드라이버간의 질질적인 입출력 통
로인 vrd_make_requeset()함수를 등록한다.

        //alloc_disk()에서 할당받은 gendisk구조체를 초기화하는 부분
        device.gd->major = VRD_DEV_MAJOR;
        device.gd->first_minor = 0;
        device.gd->fops = &vrd_fops;
        device.gd->queue = device.queue;
        device.gd->private_data = &device;//vrd_make_request()함수에서 디바이스에 대한 정보를 참조할수 있도록 private_data에 device의 주소대입
        snprintf(device.gd->disk_name, 10, "vrd");
        set_capacity(device.gd, VRD_SECTOR_TOTAL);//블록 디바이스의 총 섹터 수를 set_capacity()함수로 설정

        add_disk(device.gd);//add_disk()함수로 커널 내부에 블록디바이스를 등록

        printk(KERN_INFO "blk_device_driver init\n");
        return 0;
}
```

》 모듈을 커널에 빌드할 시 실행되는 함수

Block Device Driver

init()

```
vdisk = vmalloc(VRD_SIZE);//램디스크이기 때문에 메모리에서 해당 디바이스의 전체용량을 할당.

int state = register_blkdev(VRD_DEV_MAJOR, VRD_DEV_NAME); //블록 디바이스 드라이버를 커널에 빌드.
if(state < 0){
        printk(KERN_INFO "driver build failed..\n");
        return -1;
}
```

vmalloc(unsigned long size)

》 커널영역에서의 동적 메모리할당 함수
》 큰 메모리 buffer를 커널에서 할당
》 메모리상에서 물리적으로 연속되지 않게 할당.
》 size만큼의 버퍼를 할당

register_blkdev(unsigned int major, const char* name)

》 블록 디바이스 드라이버를 커널에 빌드.
》 /proc/devices파일에 name과 major가 저장된다.

Block Device Driver

init()

```
        device.data = vdisk;//할당한 메모리주소를 make_request()함수에서 사용할 수 있도록 device.data에 설정.
        device.gd = alloc_disk(1);//alloc_disk()함수로 블록 디바이스를 등록하기 위한 정보 구조체인 gendisk구조체를 할당받는다.
        device.queue = blk_alloc_queue(GFP_KERNEL);//vrd_make_request에서 사용하는 블록 디바이스에 필요한 요구 큐를blk_alloc_queue함수로 할당
        blk_queue_make_request(device.queue, &vrd_make_request);//blk_queue_make_request()함수를 이용해 커널과 드라이버간의 질질적인 입출력 통
로인 vrd_make_requeset()함수를 등록한다.
```

alloc_disk(unsigned long *minor)

》 gendisk구조체를 할당받기 위한 함수
》 커널 내부에서는 이 값이 1이하면 파티션이 없는 것으로 판단하여
파티션 처리를 수행하지 않는다.

blk_alloc_queue(gfp_t)

》 드라이버에서 사용할 요구 큐를 할당해주는 함수
》 gtp_t는 enum형으로 GFP_KERNEL, GFP_ATOMIC,
__GFP_HIGHMEM, __GFP_HIGH로 이루어져있음
》 해당 예제의 GFP_KERNEL는 메모리 할당이 항상 성공할 수 있게해줌.
즉, 성공할때 까지 다른 프로세스를 sleep.

Block Device Driver

init()

```
//alloc_disk()에서 할당받은 gendisk구조체를 초기화하는 부분
device.gd->major = VRD_DEV_MAJOR;
device.gd->first_minor = 0;
device.gd->fops = &vrd_fops;
device.gd->queue = device.queue;
device.gd->private_data = &device;//vrd_make_request()함수에서 디바이스에 대한 정보를 참조할수 있도록 private_data에 device의 주소대입
snprintf(device.gd->disk_name, 10, "vrd");
set_capacity(device.gd, VRD_SECTOR_TOTAL);//블록 디바이스의 총 섹터 수를 set_capacity()함수로 설정

add_disk(device.gd);//add_disk()함수로 커널 내부에 블록디바이스를 등록
```

set_capacity(struct gendisk *gd, sector_t size)

》 블록디바이스의 총 섹터수를 설정해주는 함수
》 sector_t 타입은 u64타입 (unsigned 64bit)

add_disk(struct gendisk *gd)

》 초기화된 gendisk구조체를 커널 내부에 등록해주는 함수
》 실질적으로 블록 디바이스의 파티션 검출과 등록을 해주는 함수
》 이 함수가 호출되면 커널은 블록 디바이스를 커널 내부에 등록
디바이스 파일인 /dev/vrd가 이 함수로인해 생성됨

Block Device Driver

make_request() 》요구 처리가 필요할 시 수행되는 함수

```
static unsigned int vrd_make_request(struct request_queue *q, struct bio *bio){
        vrd_device *pdevice;
        char *pVHDDData;
        char *pBuffer;
        struct bio_vec ibvec;
        struct bvec_iter iter;

        if(((bio->bi_iter.bi_sector*VRD_SECTOR_SIZE) + bio->bi_iter.bi_size) > VRD_SIZE) //bi_sector는 요구한 시작 섹터번호
                goto fail;                                                                //bi_size는 요구크기 두개의 변수가 device의 총크기를
넘어서면 오류로 처리.

        //pdevice ~ pVHDDData가 한개의 섹터가 담긴 메모리주소
        pdevice = (vrd_device*)bio->bi_disk->private_data; // 선언한 vrd_device구조체 변수의 주소를 저장하는 private_data.
                                                           // gendisk구조체를 add_disk()로 할당할때에 vrd_device의 주소가 private_data에 담김
        pVHDDData = pdevice->data + (bio->bi_iter.bi_sector*VRD_SECTOR_SIZE); // 해당 주소에서 64bit * 512byte의 주소저장.
                                                                              // data라는 필드가 즉, vrd_device의 시작주소.
        bio_for_each_segment(ibvec, bio, iter)//bio_vec구조체 벡터를 차례로 처리하기 위해 bio_for_each_segment매크로를 이용해 처리.
        {
                pBuffer = kmap(ibvec.bv_page)+ibvec.bv_offset; //page를 메모리주소로 변환 후, offset의 위치만큼 더하여 PBuffer에 실제주소 설정
                switch(bio_data_dir(bio)){//요구된 것이 읽기, 쓰기인지 알아낼 수 있는 함수.
                        case READ : memcpy(pBuffer, pVHDDData, ibvec.bv_len);
                                    break;
                        case WRITE : memcpy(pVHDDData, pBuffer, ibvec.bv_len);
                                     break;
                        default : kunmap(ibvec.bv_page);
                                  goto fail;
                }
                kunmap(ibvec.bv_page);//bvec처리가 하나 끝나면 bv_page의 매핑을 풀고
                pVHDDData += ibvec.bv_len;//pVHDDData가 가리키는 메모리의 위치를 bv_len만큼을 증가시킨다.
        }

        bio_endio(bio);
        return 0;
fail:
        bio_endio(bio);
        return 0;
}
```

Block Device Driver

make_request()

```
static unsigned int vrd_make_request(struct request_queue *q, struct bio *bio){
        vrd_device *pdevice;
        char *pVHDDData;
        char *pBuffer;
        struct bio_vec ibvec;
        struct bvec_iter iter;
```

bio struct

》연속된 Block을 묶어 놓은 세그먼트 단위로 처리하는 구조체
》기본적으로 I/O를 수행할 디스크 영역의 정보, 데이터를 저장할 메모리영역의 정보를 포함한다.
》하나의 bio구조체에는 하나의 I/O요청이 담겨져있으며 이 요청에 있는 각 블록이 bio_vec구조체에 저장되어 배열로 관리된다.

bio_vec struct

》세그먼트 입출력 Vector 구조체.
》세그먼트란 Blk I/O연산을 위한 데이터를 저장하는 '메모리' 영역 여러페이지에 걸칠 수 있다.

Block Device Driver

make_request()

```
static unsigned int vrd_make_request(struct request_queue *q, struct bio *bio){
        vrd_device *pdevice;
        char *pVHDDData;
        char *pBuffer;
        struct bio_vec ibvec;
        struct bvec_iter iter;
```

bvec_iter struct

》bio구조체에서 할당받은 세그먼트를 저장해둔 구조체

```
struct bvec_iter {
        sector_t   bi_sector;       // 요구처리가 필요한 섹터시작 번호     unsigned int
bi_size;    // 남은 I/O작업 개수
        unsigned int  bi_idx;       // ?
        unsigned int  bi_bvec_done;   // ?
}
```

Block Device Driver

make_request()

```
        if(((bio->bi_iter.bi_sector*VRD_SECTOR_SIZE) + bio->bi_iter.bi_size) > VRD_SIZE) //bi_sector는 요구한 시작 섹터번호
                goto fail;                                                                //bi_size는 요구크기 두개의 변수가 device의 총크기를
넘어서면 오류로 처리.

        //pdevice ~ pVHDDData가 한개의 섹터가 담긴 메모리주소
        pdevice = (vrd_device*)bio->bi_disk->private_data; // 선언한 vrd_device구조체 변수의 주소를 저장하는 private_data.
                                                           // gendisk구조체를 add_disk()로 할당할때에 vrd_device의 주소가 private_data에 담김
        pVHDDData = pdevice->data + (bio->bi_iter.bi_sector*VRD_SECTOR_SIZE); // 해당 주소에서 64bit * 512byte의 주소저장.
                                                                              // data라는 필드가 즉, vrd_device의 시작주소.
```

If(…) goto fail;

》 해당 요구처리가 필요한 bio구조체가 디스크의 크기보다 클 시 오류로 지정.

private_data

》 모듈빌드 시 수행되는 init()에서 vrd_device의 구조체변수의 주소를 저장해둔 포인터 변수

pVHDDData

》 해당 변수는 직접 선언해준 변수 헤더파일의 것이 아님.
》 이 변수에는 vrd_device구조체의 data필드를 이용해 disk의 첫 주소 + 하나의 섹터 주소를 저장
즉, 하나의 섹터가 해당 변수에 저장 됨.

Block Device Driver

make_request()

```
bio_for_each_segment(ibvec, bio, iter)//bio_vec구조체 벡터를 차례로 처리하기 위해 bio_for_each_segment매크로를 이용해 처리.
{
        pBuffer = kmap(ibvec.bv_page)+ibvec.bv_offset; //page를 메모리주소로 변환 후, offset의 위치만큼 더하여 PBuffer에 실제주소 설정
        switch(bio_data_dir(bio)){//요구된 것이 읽기, 쓰기인지 알아낼 수 있는 함수.
                case READ : memcpy(pBuffer, pVHDDData, ibvec.bv_len);
                            break;
                case WRITE : memcpy(pVHDDData, pBuffer, ibvec.bv_len);
                             break;
                default : kunmap(ibvec.bv_page);
                          goto fail;
        }
        kunmap(ibvec.bv_page);//bvec처리가 하나 끝나면 bv_page의 매핑을 풀고
        pVHDDData += ibvec.bv_len;//pVHDDData가 가리키는 메모리의 위치를 bv_len만큼을 증가시킨다.
}
```

bio_for_each_segment() MACRO

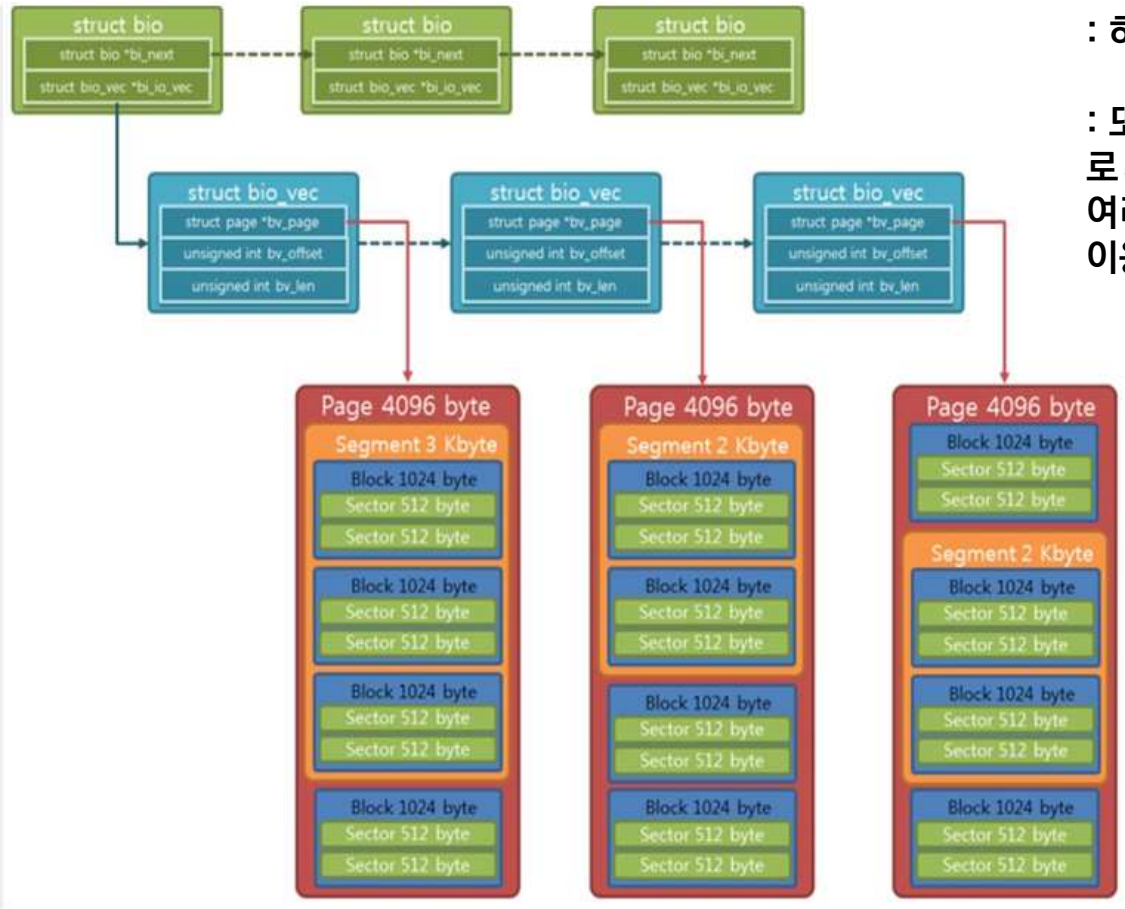
》 bio_vec구조체 배열을 전부 수행하기 위해 사용하는 매크로

bio_data_dir()

》 하나의 bio구조체가 요구하는 I/O가 무엇인지 알아낼 수 있는 함수.
》 READ, WRITE
》 default시 에러

Block Device Driver

bio struct의 구조



: 하나의 bio구조체에는 I/O요청이 하나가 존재

: 또한, bio_vec는 I/O요청이 필요한 page를 배열로 관리하는 구조체임을 알 수 있다.
여러 개의 page로 구성되어 있을 시 vector구조를 이용해 흩어져있는 page를 관리해준다.

Block Device Driver

block_device_operations

```
int vrd_open(struct block_device *bdev, fmode_t mode_t){ //블록 디바이스에 마운트 시킬시 실행되는 함수
        printk(KERN_INFO "device open\n");
        return 0;
}

void vrd_release(struct gendisk *gd, fmode_t mode_t){ //블록 디바이스를 언마운트 시킬시 실행되는 함수
        printk(KERN_INFO "device release\n");
}

int vrd_ioctl(struct block_device *bdev, fmode_t mode_t, unsigned cmd, unsigned long error){
        return -ENOTTY; //램디스크는 ioctl의 명령에 모두 -ENOTTY를 반환하여 처리하면 됨
                        //ioctl()함수는 대부분 커널 내부에서 알아서 처리한다.
}

static struct block_device_operations vrd_fops = { //파일오퍼레이션
        .owner = THIS_MODULE,
        .open = vrd_open,
        .release = vrd_release,
        .ioctl = vrd_ioctl
};
```

》 디바이스파일의 시스템 콜이 일어날 시 수행되는 함수들을 매핑

open()
》 해당 함수는 디바이스파일이 마운트 될 시 수행되는 함수
release()
》 해당 함수는 디바이스파일이 언마운트 될 시 수행되는 함수
ioctl()
》 해당 함수는 디바이스파일의 입출력 제어가 필요할 시 수행되는 함수

Block Device Driver

exit()

```
void vrd_exit(void){
        del_gendisk(device.gd);//gendisk구조체를 커널에서 제거
        put_disk(device.gd);//할당된 gendisk구조체의 메모리를 해제
        unregister_blkdev(VRD_DEV_MAJOR, VRD_DEV_NAME);//블록 디바이스를 제거
        vfree(vdisk);//램디스크의 전체크기를 위해 할당한 메모리를 반환
        printk(KERN_INFO "blk_device_driver exit");
}

module_init(vrd_init);
module_exit(vrd_exit);

MODULE_LICENSE("GPL");
```

del_gendisk()

》 커널에 할당된 gendisk구조체를 커널에서 제거
》 디바이스 파일이 자동삭제됨 add_disk()함수와 반대역할

put_disk()

》 커널에 할당된 gendisk구조체의 메모리를 해제

unregister_blkdev()

》 커널에 빌드된 블록디바이스 드라이버를 제거
》 /proc/devices파일에서 해당 드라이버정보가 제거됨

Block Device Driver

오브젝트(.ko) 파일 생성 및 의존성 확인

```
lyc@lyc-linux:~/pi/device_driver/test_blk_driver/ram_disk$ make
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- -C/home/lyc/pi/linux M=/home/lyc/pi/device_driver/test_blk_driver/ram_disk modules
make[1]: 디렉터리 '/home/lyc/pi/linux' 들어감
  CC [M]  /home/lyc/pi/device_driver/test_blk_driver/ram_disk/vrd.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/lyc/pi/device_driver/test_blk_driver/ram_disk/vrd.mod.o
  LD [M]  /home/lyc/pi/device_driver/test_blk_driver/ram_disk/vrd.ko
make[1]: 디렉터리 '/home/lyc/pi/linux' 나감
lyc@lyc-linux:~/pi/device_driver/test_blk_driver/ram_disk$ ls
Makefile  Makefile_for_x86  Module.symvers  modules.order  vrd.c  vrd.ko  vrd.mod.c  vrd.mod.o  vrd.o
lyc@lyc-linux:~/pi/device_driver/test_blk_driver/ram_disk$ file vrd.ko
vrd.ko: ELF 32-bit LSB relocatable, ARM, EABI5 version 1 (SYSV), BuildID[sha1]=12680c2197da984f8fdb307edd5590a2d7c12a83, not stripped
lyc@lyc-linux:~/pi/device_driver/test_blk_driver/ram_disk$
```

$make

》arm머신의 오브젝트 파일을 생성하는 Makefile에 따라 컴파일 실행

$file vrd.ko

》생성된 오브젝트 파일의 의존성 확인

Block Device Driver

Target으로 오브젝트 파일 전송 및 원격 접속 실행

```
lyc@lyc-linux:~/pi/device_driver/test_blk_driver/ram_disk$ scp ./vrd.ko pi@192.168.10.53:/home/pi
The authenticity of host '192.168.10.53 (192.168.10.53)' can't be established.
ECDSA key fingerprint is SHA256:CuvpjkPEm/G9GWS+uCCYK93beWP8970qMJLaQ5Uxqes.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.10.53' (ECDSA) to the list of known hosts.
pi@192.168.10.53's password:
vrd.ko                                                                    100% 7292    508.1KB/s   00:00
lyc@lyc-linux:~/pi/device_driver/test_blk_driver/ram_disk$ ssh pi@192.168.10.53
pi@192.168.10.53's password:
Linux raspberrypi 4.14.82-v7+ #1 SMP Thu Nov 22 20:57:19 KST 2018 armv7l

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Thu Jan 10 06:26:30 2019
pi@raspberrypi:~ $ ls
Desktop  Documents  Downloads  MagPi  Music  Pictures  Public  Templates  Videos  vrd.ko
```

$scp

》Target의 Home dir로 오브젝트 파일 전송

$ssh

》Target의 터미널로 원격접속

》호스트명이 pi로 바뀌는 것을 확인할 수 있다.

Block Device Driver

Target보드에 모듈적재 시키기

```
pi@raspberrypi:~ $ sudo insmod vrd.ko
pi@raspberrypi:~ $ lsmod
Module                  Size  Used by
vrd                    16384  0
rfcomm                 49152  4
bnep                   20480  2
fuse                  110592  3
hci_uart               36864  1
btbcm                  16384  1 hci_uart
serdev                 20480  1 hci_uart
bluetooth             364544  29 hci_uart,bnep,btbcm,rfcomm
ecdh_generic           28672  1 bluetooth
brcmfmac              307200  0
brcmutil               16384  1 brcmfmac
cfg80211              569344  1 brcmfmac
rfkill                 28672  6 bluetooth,cfg80211
snd_bcm2835            32768  1
snd_pcm                98304  1 snd_bcm2835
snd_timer              32768  1 snd_pcm
snd                    69632  5 snd_timer,snd_bcm2835,snd_pcm
uio_pdrv_genirq        16384  0
fixed                  16384  0
uio                    20480  1 uio_pdrv_genirq
joydev                 20480  0
evdev                  24576  4
i2c_dev                16384  0
ip_tables              24576  0
x_tables               32768  1 ip_tables
ipv6                  430080  24
```

$sudo insmod vrd.ko

》Kernel에 디바이스 드라이버 모듈 적재
》init()함수에 정의해둔 소스코드를 실행한다.
》디바이스 파일인 /dev/vrd가 생성되며
/proc/devices파일에 드라이버 정보가 추가된다.

$lsmod

》현재 커널에 적재된 모듈을 확인할 수 있다.
》vrd라는 모듈이 적재되었음을 확인가능

Block Device Driver

/proc/devices

```
Block devices:
  1 ramdisk
  7 loop
  8 sd
 65 sd
 66 sd
 67 sd
 68 sd
 69 sd
 70 sd
 71 sd
128 sd
129 sd
130 sd
131 sd
132 sd
133 sd
134 sd
135 sd
179 mmc
240 vrd
259 blkext
```

$cat /proc/devices

》현재 커널에 적재된 디바이스 드라이버들을 확인가능

》왼쪽이 major num이며 오른쪽이 dev name이다.

》예제의 vrd는 major가 240번임을 확인할 수 있다.

vrd dev의 정보확인

```
Disk /dev/vrd: 4 MiB, 4194304 bytes, 8192 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
```

$sudo fdisk -l(소문자 L)

》시스템의 디스크와 파티션들을 출력해주는 명령어와 옵션.

》해당 명령어로 vrd디바이스의 정보를 확인가능 하다.

Block Device Driver

device에 fs올리기

```
pi@raspberrypi:~ $ sudo mkfs -t ext4 /dev/vrd
mke2fs 1.43.4 (31-Jan-2017)
Creating filesystem with 4096 1k blocks and 1024 inodes

Allocating group tables: done
Writing inode tables: done
Creating journal (1024 blocks): done
Writing superblocks and filesystem accounting information: done
```

$sudo mkfs -t ext4 /dev/vrd

》 vrd dev에 ext4 파일 시스템을 올리는 명령어

》 정상적으로 수행되는 것을 확인가능.

마운트를 위한 dir생성

```
pi@raspberrypi:~ $ sudo mkdir /mnt/testdir
pi@raspberrypi:~ $ ls -l /mnt
total 4
drwxr-xr-x 2 root root 4096 Jan 11 03:39 testdir
pi@raspberrypi:~ $ ls -l /mnt/testdir
total 0
```

$sudo mkdir /mnt/testdir

》 mount dir에 testdir라는 dir를 생성

》 mount dir임으로 root권한으로 생성해야 한다.

》 생성 후 testdir의 내용을 확인하면 아무것도 없는것을 확인가능

Block Device Driver

디바이스를 마운트시키기

```
pi@raspberrypi:~ $ sudo mount /dev/vrd /mnt/testdir
pi@raspberrypi:~ $ ls -l /mnt/testdir
total 12
drwx------ 2 root root 12288 Jan 11 03:39 lost+found
pi@raspberrypi:~ $ touch testfile
pi@raspberrypi:~ $ sudo mv testfile /mnt/testdir
pi@raspberrypi:~ $ ls -l /mnt/testdir
total 12
drwx------ 2 root root 12288 Jan 11 03:39 lost+found
-rw-r--r-- 1 pi   pi        0 Jan 11 03:57 testfile
pi@raspberrypi:~ $ sudo rm /mnt/testdir/testfile
pi@raspberrypi:~ $ ls -l /mnt/testdir
total 12
drwx------ 2 root root 12288 Jan 11 03:39 lost+found
```

$sudo mount /dev/vrd /mnt/testdir

》 디바이스를 해당 dir에 마운트

》 vrd devic의 마운트 포인트는 /mnt/testdir가 된다.

》 드라이버의 open()함수가 실행된다.

》 빈 파일이나 4Mb이하의 파일들은 정상적으로 옮겨 갈 수있다.

》사진 내의 빈파일인 testfile이 mv명령어로 옮겨가는 것을 확인가능하다.(rm로 삭제되는 것도 확인가능)

```
pi@raspberrypi:~ $ lsmod
Module                  Size  Used by
vrd                    16384  1
```

$lsmod

》 현재 커널에 빌드되어 있는 모듈정보를 출력

》 마운트되어 있는 vrd로 인해 Used by란이 1로 증가 된 것을 확인 가능하다. (초기값 0)

Block Device Driver

디바이스를 언마운트시키기

```
pi@raspberrypi:~ $ sudo rmmod vrd
rmmod: ERROR: Module vrd is in use
pi@raspberrypi:~ $ sudo umount /dev/vrd
pi@raspberrypi:~ $ sudo rmmod vrd
```

$sudo umount /dev/vrd

》 디스크등의 디바이스들을 언마운트 시키는 명령어
》 언마운트를 시키기 전에 rmmod명령을 사용할 시
vrd모듈을 사용하고 있음으로 제거 불가능 에러가 뜬다.
》 그럴시에는 해당 디바이스를 언마운트를 실시한다.
》 lsmod명령어 수행시 vrd모듈의 Used by란이 1감소함

$sudo rmmod vrd

》 커널에 빌드된 모듈을 언빌드 시키는 명령어
》 vrd모듈을 언빌드 시킨다.

Block Device Driver

로그확인

```
[   79.220197] vrd: loading out-of-tree module taints kernel.
[   79.224579] blk_device_driver init
[   79.227492] device open
[   79.231806] device release
[  228.931273] device open
[  228.931304] device release
[  228.932131] device open
[  228.933597] device release
[  228.934043] device open
[  228.934348] device release
[  228.936291] device open
[  228.936309] device release
[  228.936332] device open
[  228.936353] device release
[  228.936499] device open
[  228.936514] device release
[  228.936530] device open
[  228.936543] device release
[  228.936630] device open
[  228.936816] device release
[  228.936964] device open
[  228.937116] device release
[  228.937148] device open
[  228.959022] device release
[  254.052733] device open
[  254.054494] device release
[  254.054816] device open
[  254.055831] EXT4-fs (vrd): mounted filesystem with ordered data mode. Opts: (null)
[  309.870847] device release
[  315.745344] blk_device_driver exit
```

$dmesg

〉〉 시스템 로그를 확인가능하다.

〉〉 로그를 확인하여 드라이버의 작동을 확인가능하다.

E N D