시스템프로그래밍
1차 과제

Nilfs , Ext 4 파일시스템 분석

15 조
2016320272 권영진
2016320265 박현종
Free Day 사용일수 : 4일

목차

1-1. 역할 배분

권영진 :

- (블락번호, 파일시스템이름, 시간) 위치 파악 및 코드작성
- Loadable Kernal Module & Proc 파일 코드 작성
- 배경지식 보고서 담당

박현종 :

- blk-core.c 의 원형큐 구조체 코드 작성
- submit_bio() 함수의 코드수정
- 코드분석 보고서 담당

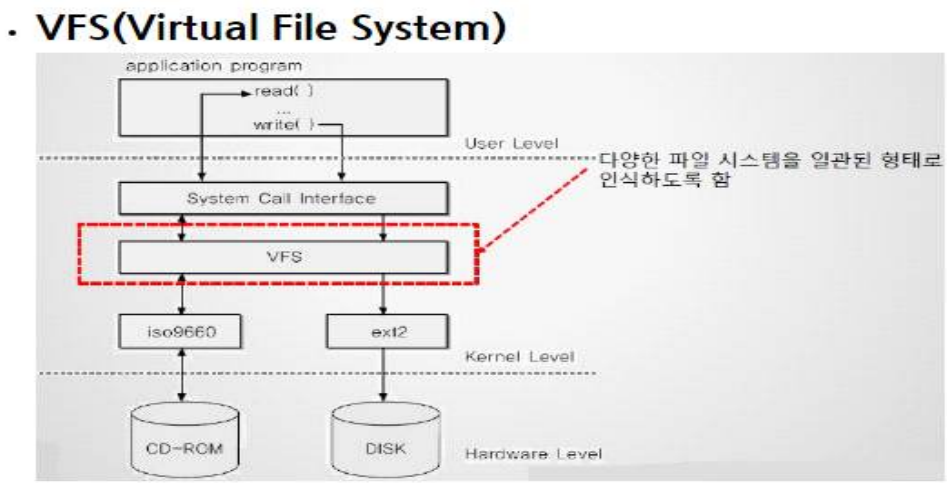
1-2. 개발 환경

가상머신 : Oracle Virtual Box
운영체제 : Ubuntu 16.04.6 LTS (64bit)
커널 : linux-4.4
파일시스템 : nilfs, ext4
하드웨어 스펙 : Intel(R) Core(TM) i5-7500 CPU @ 3.40GHz (CPU 4 core), 8GB(RAM)
가상머신 할당스펙 : CPU 4 core, 4GB RAM
사용언어 : C

1-3. 배경지식

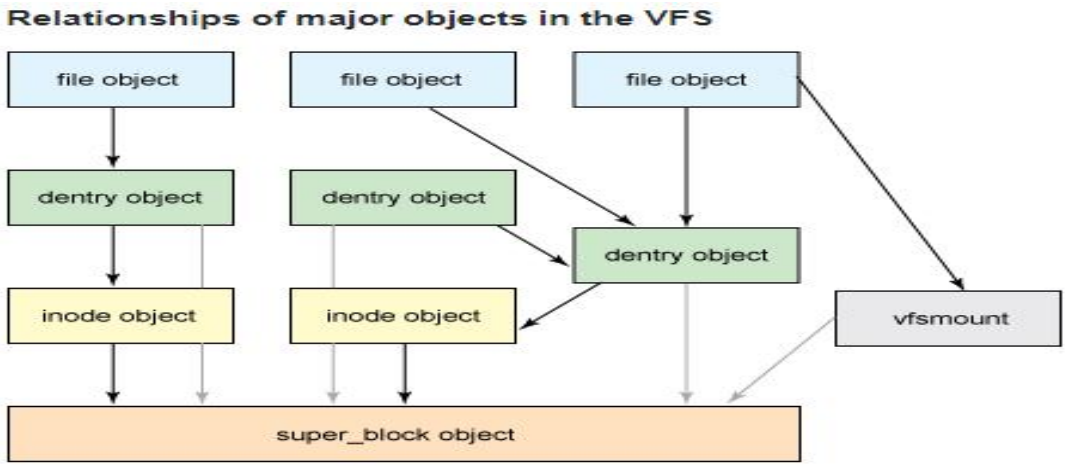
- VFS(Virtual File System)

VFS는 사용자가 다양하게 존재하는 파일 시스템에 대해 동일한 방법으로 접근이 가능하도록 설계된 실제 파일 시스템 위의 추상 계층이다. 즉, 사용자 프로세스는 VFS를 통해 파일 시스템이나 물리 매체의 종류와 관계없이 open, read, write 등과 같은 시스템 콜을 사용할 수 있게 된다.



superBlock : 마운트 된 파일 시스템에 대한 정보를 저장, 전체 파일 시스템을 나타냄
inode : 특정 파일에 대한 일반 정보를 저장. 디스크에 저장한 파일 제어 블록

file : 열린 파일과 프로세스 사이의 상호 작용과 관련한 정보를 저장. 각 프로세스가 열린 파일을 가지는 동안 커널 메모리에만 존재
dentry : 디렉토리 항목(특정 파일 이름과 아이노드)과 이에 대응하는 파일의 연결에 대한 정보를 저장.



- Block Layer

블록 장치, 즉 하드 디스크는 CPU는 물론 메모리에 비해서도 동작 속도가 현저히 느리다. 이는 전통적인 방식의 디스크에서 디스크 헤드가 원하는 위치까지 이동하기 위해 필요한 시간(seek time)이 매우 길기 때문이다. 따라서 리눅스의 VFS 계층은 디스크 접근을 최소화하기 위해 페이지 캐시를 이용하여 한 번 접근한 디스크의 내용을 저장해 둔다. 하지만 여기서는 이러한 페이지 캐시의 작용은 건너뛰고 실제로 블록 장치와 I/O 연산만을 고려하자.

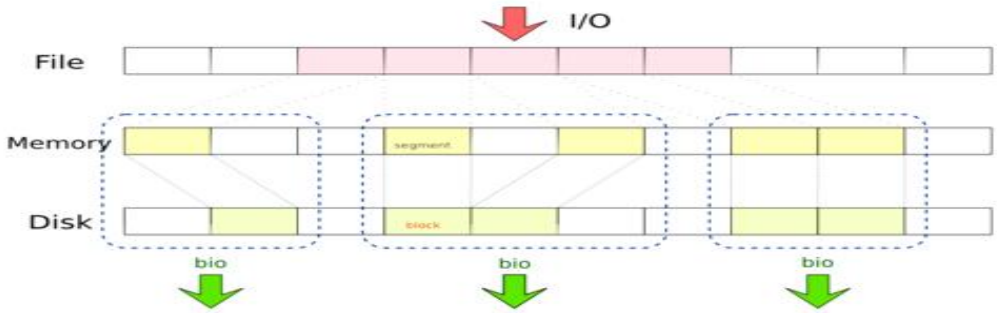
블록 I/O의 시작 지점은 submit_bio() 함수이다. 일반적인 파일 시스템의 경우 buffer_head (bh)라는 구조체를 통해 디스크 버퍼를 관리하는데 이 경우 submit_bh() 함수가 사용되지만 이는 bh의 정보를 통해 bio 구조체를 할당하여 적절히 초기화한 후 다시 submit_bio() 함수를 호출하게 된다. 이 함수는 I/O 연산의 종류 (간단하게는 READ 혹은 WRITE) 및 해당 연산에 대한 모든 정보를 포함하는 bio 구조체를 인자로 받는다. bio 구조체는 기본적으로 I/O를 수행할 디스크 영역의 정보와 I/O를 수행할 데이터를 저장하기 위한 메모리 영역의 정보를 포함한다. 여기서 몇가지 용어가 함께 사용되는데 혼동의 여지가 있으므로 간략히 정리하고 넘어가기로 한다.

섹터(sector)라는 것은 장치에 접근할 수 있는 최소의 단위이며 (H/W적인 특성) 대부분의 장치에서 512Byte 이므로, 리눅스 커널에서는 섹터는 항상 512 Byte으로 가정하여 sector_t 타입은 512Byte 섹터 단위 크기를 나타낸다.

블록(block)은 장치를 S/W적으로 관리하는 (즉, 접근하는) 크기로 섹터의 배수이다. 일반적으로 파일 시스템 생성 (mkfs) 시 해당 파일 시스템이 사용할 블록 크기를 결정하게 되며 현재 관리의 용이성을 위해 블록 크기는 페이지 크기 보다 크게 설정될 수 없다. 즉, 일반적인 환경에서 블록의 크기는 512B, 1KB, 2KB, 4KB 이다. 하나의 블록은 디스크 상에서 연속된 섹터로 이루어진다.

세그먼트(segment)는 장치와의 I/O 연산을 위한 데이터를 저장하는 "메모리" 영역을 나타내

는 것으로 일반적으로는 페이지 캐시 내의 일부 영역에 해당 할 것이다. 하나의 블록은 메모리 상에서 동일한 페이지 내에 저장되지만 하나의 I/O 연산은 여러 블록으로 구성될 수도 있으므로 하나의 세그먼트는 (개념적으로) 여러 페이지에 걸칠 수도 있다.
블록 I/O 연산은 기본적으로 디스크에 저장된 데이터를 메모리로 옮기는 것 (READ) 혹은 메모리에 저장된 데이터를 디스크로 옮기는 것 (WRITE)이다. I/O 연산이 여러 블록을 포함하는 경우 약간 복잡한 문제가 생길 수 있는데 이러한 블록 데이터가 디스크 혹은 메모리 상에서 연속되지 않은 위치에 존재할 수 있기 때문이다. 따라서 bio 구조체는 이러한 상황을 모두 고려하여 I/O 연산에 필요한 정보를 구성한다.



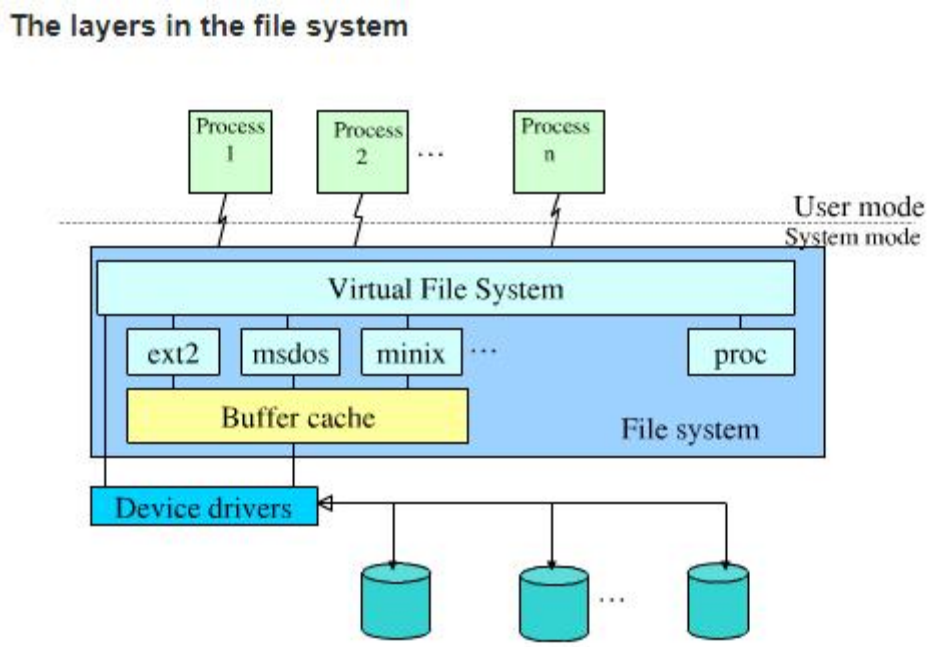
- Proc File System

proc파일 시스템은 운영체제의 각종 정보를 커널모드가 아닌 유저모드에서 쉽게 접근할 수 있도록 만들어 줌으로 시스템 정보를 일반 프로그래머가 쉽게 접근 할 수 있도록 도와준다.
리눅스에서는 프로세스 정보뿐만 아닌 다른 시스템 정보들까지 제공해 준다. 실제 프로세스 상황감시, CPU 사용률, 인터럽트, 네트워크 패킷전송량, 적재된 모듈, CPU정보등의 데이터를 어렵지 않게 얻어 올 수 있다.
일반적으로 사용되는 파일 시스템은 상당한 오버헤드를 가지고 있다. 각 파일의 inode와 superblocks와 같은 객체를 관리해야 하며 이러한 정보를 필요할 때마다 운영체제에 요청해야 한다. 이들 파일시스템의 데이터들은 단현화 현상이 발생할 수도 있다. 운영체제는 이러한 모든 것을 관리해 주어야 하며, 당연히 상당한 오버헤드가 발생하게 된다. proc파일 시스템은 이러한 일반 파일시스템의 문제점을 없애기 위해서 리눅스 커널에서 직접 파일시스템을 관리하는 방법을 채택하고 있다.
이 proc은 물리적인 파일시스템 장치를 필요하지 않아 임베디드 시스템을 설계할 때 요긴하게 사용되며 커널 모듈 프로그램이에 사용된다. 커널 모듈 프로그램은 주로 장치를 올리기 위하여 사용되는데, 커널 레벨에서 동작하는 것은 사용자가 모듈에대한 정보를 아는 것이 어렵다. 일반 파일시스템의 IPC를 이용하면 에로사항이 발생할 수 있으므로 proc을 사용한다.

- EXT4 (Extended File System 4)

리눅스 초기 ext부터 시작하여 발전을 걸쳐서 만들어진 ext3 파일시스템을 확장한 파일시스템으로 Extent라는 기능을 제공하여, 파일에 디스크 할당 시 물리적으로 연속적인 블록을 할당할 수 있도록 하여, 파일 접근 속도 향상 및 단편화를 줄이도록 설계된 파일시스템이다.

리눅스는 모든파일 시스템은 inode를 통해 파일을 관리하며 디렉토리는 단순히 파일의 목록



을 가지고있는 파일이다. ext4의 경우는 효율적인 디스크 할당을 위하여 저장장치를 블록그룹으로 구분하였다. 블록그룹은 4KB가 일반적이며 변경가능하다.

ext4는 48비트 내부 주소 지정을 사용하여 최대 16TB의 파일을 파일시스템 1,000,000TiB(1 EiB)까지 할당하는 것이 이론적으로 가능하다. 일부 사용자 환경 유틸리티에 의해 초기 ext4는 16TiB파일 시스템으로 제한되었지만, 2011년부터 16TiB 이상의 ext4 파일 시스템 생성을 지원한다. Red Hat Enterprise Linux는 공식적으로 ext4 파일 시스템을 최대 50TiB까지만 지원하며, 100TiB이하의 ext4 볼륨을 권장한다.

[그림 7-3] ext4 파일 시스템의 구조

- NILFS2(New Implementation of a Log-structured File System)

NILFS는 저장 매체가 원형 버퍼처럼 취급되고 새로운 블록이 항상 끝까지 기록된다는 점에서 LFS 파일 시스템이다. LFS(Log structured Filesystem)는 wear-leveling을 수행하기 때문에 플래시 미디어에 자주 사용됩니다. 빠른 쓰기 및 복구 시간을 가지고 있으며 하드웨어

오류시 파일 데이터 및 시스템 일관성에 대한 최소손상을 보장합니다. nilfs2에서는 지속적인 스냅 샷을 제공하는데, 사용자는 최근에 실수로 덮어 쓰건 삭제한 파일을 복원할 수 있다.

- LFS (Log Structured File System)

데이터를 업데이트 할 때 전에 기록한 데이터는 무효화 하고 새로 쓰여지는 데이터는 다른 곳에 기록하며 순차적으로 쓴다. 이러한 순차적 기록방식은 플래시 메모리 외에도 하드디스크와 같은 대부분의 장치에서 임의쓰기성능보다 나은 성능을 보여주기때문에 로그구조 파일시스템을 많이 이용한다. 또한 로그 구조 파일시스템은 데이터를 로그에 순차적으로 기록하여 관리하기 때문에 쓰기 성능과 복구에 뛰어나다. 기존 파일 시스템이 메타 데이터 영역과 유저 데이터 영역을 나누어 관리하였던 것이 비해, 로그 구조 파일시스템은 각 데이터를 모두 로그에 기록한다.

2-1. 코드분석

[blk-core.c] : 원형큐 구현

```
struct circular_queue {

unsigned long long bk_buf[2000];
int idx = 0;
long long int tm_buf[2000];
const char* tp_buf[2000];

}cq;

EXPORT_SYMBOL(cq);

struct timeval mytime;
```

ext4와 nilfs2 모두 write 수행시 blk-core.c의 submit_bio 함수를 거치게 된다. 이 함수에서 디스크의 정보를 얻을 수있기 때문에 큐를 전역변수로 선언하고 submit_bio함수내에서 디스크의 정보들을 큐에 삽입하게 만들었다.

필요한 정보들인 디스크의 섹터정보 , write가 수행된 time , write가 수행되고 있는 파일시스템의 type을 저장하는 배열을 묶어서 circular_queue라는 구조체로 만든다.
배열들의 크기는 2000으로 하고 순환큐를 구현하기위해서 2000이 넘으면 변수 idx의 모듈러 연산을 통해 순환되도록 만들었다. kernel module인 hw1.c에서 큐를 사용할수있도록 EXPORT_SYMBOL을 해준다.

[blk-core.c] : submit_bio 구현

```
                if (rw & WRITE) {
                        count_vm_events(PGPGOUT, count);

                if(bio->bi_iter.bi_sector!=NULL){
                    cq.bk_buf[cq.idx] = (unsigned long long) bio-
>bi_iter.bi_sector;

                    do_gettimeofday(&mytime);

                    cq.tm_buf[cq.tm_buf] = (unsigned long long)(mytime.tv_sec) *
1000000 + (unsigned long long)(mytime.tv_usec);

                                    if(bio->bi_bdev != NULL) if(bio->bi_bdev
>bd_super != NULL) if(bio->bi_bdev->bd_super->s_type != NULL)
                                        cq.tp_buf[cq.idx] = bio->bi_bdev
>bd_super->s_type->name;

                    cq.idx = (cq.idx + 1) % 2000;
                    }
```

디스크의 섹터정보는 bio->bi_iter를 통해 bio구조체인 bi_sector에 접근하여 알수있었다.
bi_sector에 접근할때는 kernel crash를 방지하기위해 null인지 먼저 체크를 해준다.

write가 수행된 시간을 알기위해 do_gettimeoofday라는 현재시간을 구해주는 함수를 사용했다. timeval 구조체의 tv_usec이라는 변수에 microsecond단위로 시간이 저장되기때문에 second단위로 시간을 측정해주는 tv_sec에 1000000을 곱해 더해주는 식으로 했다.

write가 수행되고있는 파일시스템의 type을 알기위해 bio->bi_bdev , bio->bi_bdev->bd_super , bio->bi_bdev->bd_super->s_type 순서로 접근하여 NULL인지 판단하는 예외처리를 해준다.

쓰기가 발생하는 순간 앞에서 선언한 큐에 각 정보를 입력한다. 이때 순환큐를 만드기위해 모듈러 연산을 해준다.

[seg_buf.c]

```
        bio->bi_end_io = nilfs_end_bio_write;
        bio->bi_private = segbuf;

        if(bio->bi_bdev!=NULL)
                bio->bi_bdev->bd_super = segbuf->sb_super;

        submit_bio(mode, bio);
        segbuf->sb_nbio++;

        wi->bio = NULL;
        wi->rest_blocks -= wi->end - wi->start;
        wi->nr_vecs = min(wi->max_pages, wi->rest_blocks);
```

bio->bi_bdev->bd_super = segbuf->sb_super 이 부분을 submit_bio함수가 호출되기전에 넣어줌으로써 각 블록이 superblock에대한 정보를 가질수있게 한다.

[hw1.c] : proc & lkm

```
#define PROC_DIRNAME "myproc"
#define PROC_FILENAME "hw1"

static struct proc_dir_entry *proc_dir;
static struct proc_dir_entry *proc_file;

extern struct circular_queue cq;

static int hw1_open(struct inode *inode, struct file *file) {
    printk(KERN_ALERT "<hw1_file_open>\n");
    return 0;
}

static ssize_t hw1_write(struct file *file, const char __user *user_buffer,
size_t count, loff_t *ppos) {
    int i, start_index;
    struct file *filp;
    char tmp[19];

    start_index = cq.idx;


    mm_segment_t old_fs = get_fs();
    set_fs(KERNEL_DS);

    //open file
    filp = filp_open("/tmp/result.csv", O_WRONLY|O_CREAT, 0644);
    if(IS_ERR(filp)) {
        printk("<file_open_error>\n");
            set_fs(old_fs);

            return count;
    }
    else {
        printk("<file_open_success>\n");
    }

    printk(KERN_ALERT "<hw1_write>\n");


    for(i = start_index + 1; i != start_index; i++) {
        i%=2000;

                if(cq.tp_buf[i])
                if(user_buffer)
                if(strncmp(cq.tp_bif[i], user_buffer, count-1) == 0)
                if(cq.bk_buf[i]!=0) {
                        printk(KERN_INFO "%d:[%lld] %lld\n", i, cq.tm_buf[i],
cq.bk_buf[i]);

                    snprintf(tmp, 19, "%lld", cq.tm_buf[i]);
                    vfs_write(filp, tmp, strlen(tmp), &filp->f_pos);
                    vfs_write(filp, ", ", 2, &filp->f_pos);

                    snprintf(tmp, 19, "%lld", cq.bk_buf[i]);

                    vfs_write(filp, tmp, strlen(tmp), &filp->f_pos);
                    vfs_write(filp, ", ", 2, &filp->f_pos);

                    printk("%s\n", cq.tp_buf[i]);

                    snprintf(tmp, 19, "%s", cq.tp_buf[i]);
                    vfs_write(filp, tmp, strlen(tmp), &filp->f_pos);
                    vfs_write(filp, "\n", 1, &filp->f_pos);

                    cq.tm_buf[i]=0;
                    cq.bk_buf[i]=0;
                    cq.tp_buf[i]=NULL;
            }
   }

    printk(KERN_ALERT "<hw1_write_complete>\n");

        //close file
    filp_close(filp, NULL);

    //restore kernel memory setting
    set_fs(old_fs);
```

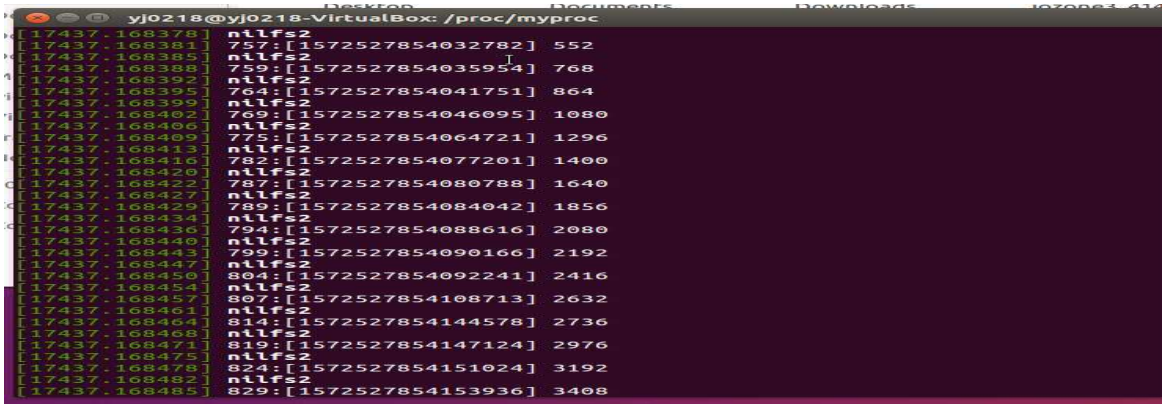
```
    return count;
}

static const struct file_operations hw1_proc_fops = {
    .owner = THIS_MODULE,
    .open = hw1_open,
    .write = hw1_write,
};

static int __init hw1_init(void) {
    printk(KERN_ALERT "<hw1_module_up>\n");
    proc_dir = proc_mkdir(PROC_DIRNAME, NULL);
    proc_file = proc_create(PROC_FILENAME, 0600, proc_dir, &hw1_proc_fops);
    return 0;
}

static void __exit hw1_exit(void) {
    printk(KERN_ALERT "<hw1_module_down>\n");

    return;
}

module_init(hw1_init);
module_exit(hw1_exit);

MODULE_AUTHOR("kyj,phj");
MODULE_DESCRIPTION("System Programming");
MODULE_LICENSE("GPL");
MODULE_VERSION("NEW");
```

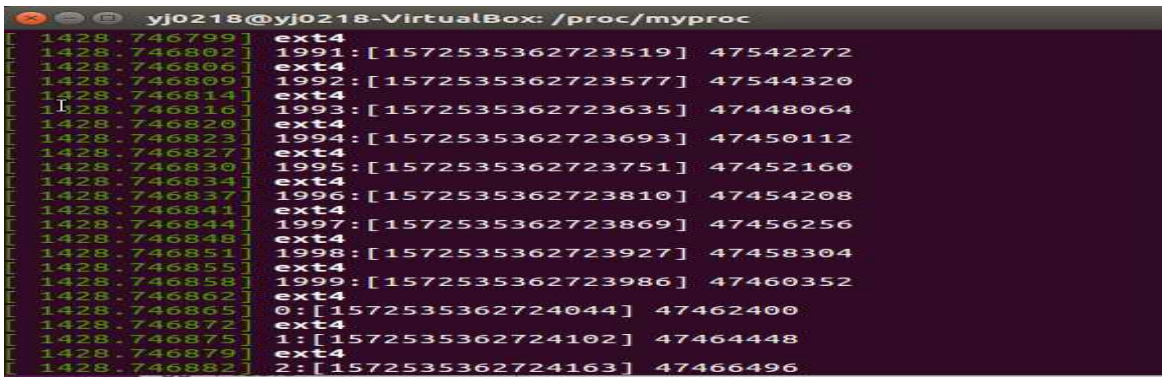
hw1 모듈에 write가 발생하면 버퍼값에 접근하기전에 set_fs 함수를 통해 파일시스템을 설정해준다. filp_open을 통해 쓰고자하는 파일인 /tmp/result.csv 파일을 열어 파일구조체로 만들어 주고 IS_ERR함수로 예외처리를 해준다. 큐의 변수들이 비어있지 않고 strcmp함수를 통해 큐에 써져 있는 파일시스템명과 사용자로부터 입력받은 파일시스템명을 비교해서 일치함이 판명되면 /tmp/result.csv 파일을 열어 버퍼의 값들을 입력해준다.

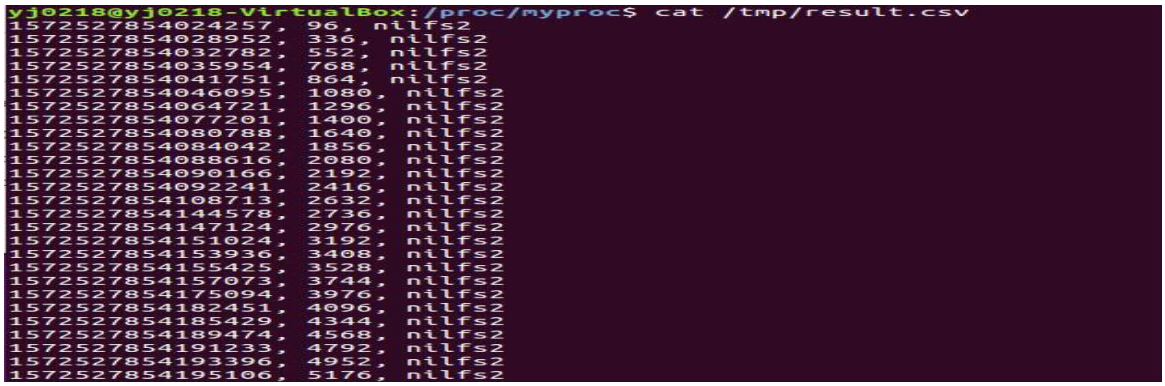
2-2. 실행 방법

1. readme.txt 에 따라 block.c, hw1.c, seg_buf.c 적절한 위치에 삽입.
2. 커널 컴파일 진행 (make -> make install)
3. .ko 파일 모듈을 커널에 삽입. (insmod)
4. Nilfs, ext4 각각의 폴더 생성후 파일시스템에 따라 마운트 (mount)
5. 각 파일 시스템에서 iozone -a -i 0을 통해서 write연산 수행 (iozone)
6. /proc/myproc에 생성된 hw1 파일에 권한을 주고 echo를 통해 버퍼내용 출력 (chmod 777)
7. 최종적으로 .csv 파일이나 dmesg 를 통해서 출력정보 확인

[dmesg & .csv]

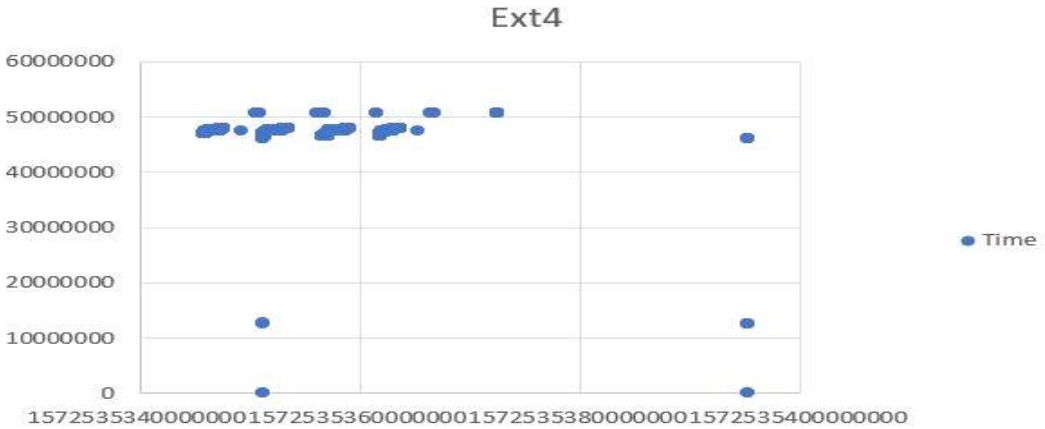






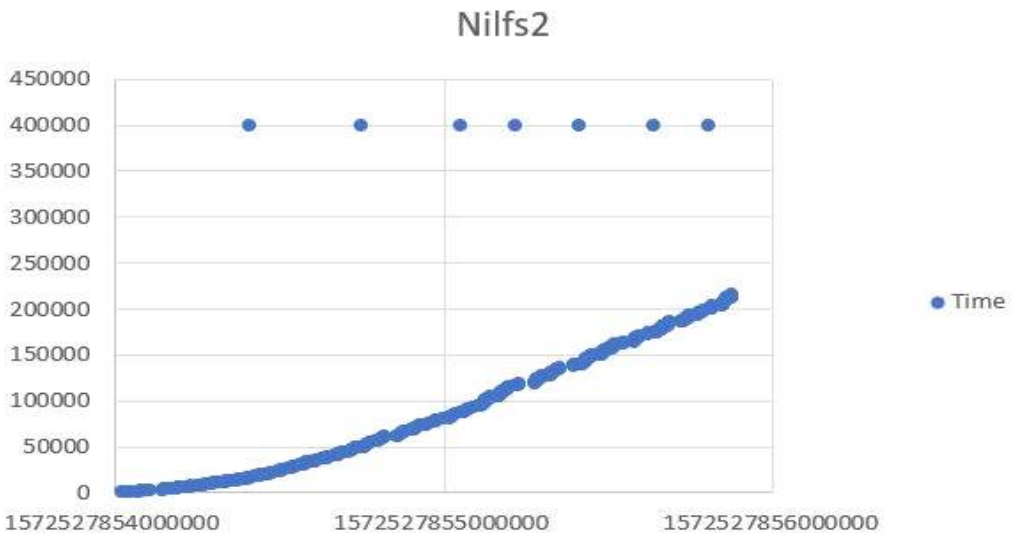
3-1. 결과분석

[Ext 4 Graph]



ext4 파일에서는 write연산에서 inode의 위치에 뜨른 파일, 디렉토리가 순차적인 증가가 아닌 흩어져서 나타난다. 즉 파일구조가 super block, inode, data bock을 디스크에서 다른 위치에 저장한뒤 맵핑을 하여 참조하기 때문이다. 그렇기 때문에 쓰기에 오랜 시간이 걸릴 수 있다.

[Nilfs 2 Graph]



nilfs2에서 write연산은 시간이 흐름에 따라 block number가 상승하고 있다. nilfs의 write, rewrite 연산이 LFS의 구조처럼 inode를 append 시키는 과정이라고 생각하면되는데. 즉 쓰기연산에 있어서 연속된 섹터에 순차적으로 접근하는 것이다. 따라서 상승곡선을 그리고 있다.

3-2. 에로사항

1. if (참조 != Null)
 컴파일 과정에서 crash가 발생하여 파일 시스템의 섹터와 bio의 슈퍼블락이 nilfs의 슈퍼블락 포인터를 참조할 때 등 모든 참조 부분에 대하여 예외처리를 주었다.

2. undefiend SYMBOL
 처음 blk-core.c 파일에서 버퍼를 만들어 EXPORT SYMBOL을 하였는데도 모듈을 make하면 undefined symbol 에러가 발생하였다. 이후 symbol table이 첫 커널 컴파일에 생성됨을 인지하고 커널 컴파일을 진행하여 모듈 컴파일을 진행할 수 있었다.

3. chmod 777
 생성된 .ko 모듈을 삽입 후 이제 버퍼의 내용을 proc파일에다 쓰려 하는데 권한 문제로 echo명령어를 사용할 수 없었다. 권한을 chmod 명령어를 통해 변경하여 수행, 그 이후 문제없이 출력 되었다.