시스템 프로그래밍 1 차 과제 보고서

제출자 2019320069 정채운 : LKM 코드 작성, 보고서 작성

2019320122 이권은 : 커널 코드 수정, 보고서 작성

제출일자 2021 년 11 월 2 일

Freeday 사용일수 0 일

개발환경

Virtual machine: Oracle VM Virtualbox 6.1.18

Guest OS: Ubuntu 16.04 LTS

Linux kernel version: 4.4.0

사용언어: C

하드웨어 스펙: Intel® Core™ i7-2600 CPU 3.40 GHz, RAM 8.00GB, HDD 1TB (SAMSUNG HD103SJ)

배경 지식 설명

- VFS

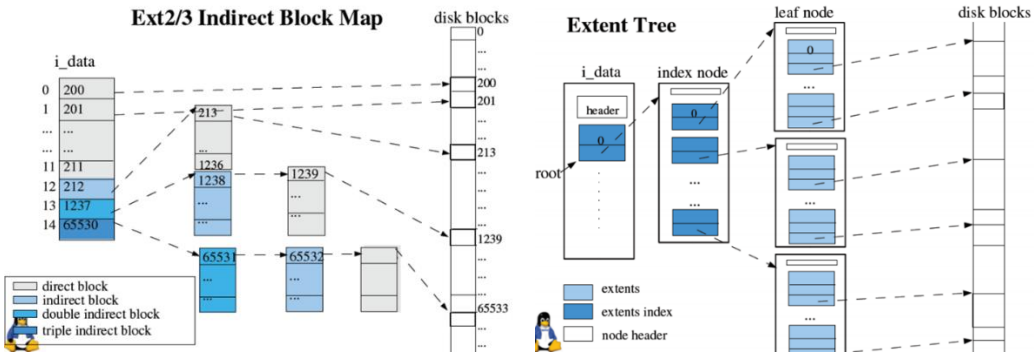
VFS 는 시스템 콜을 파일 시스템이나 물리 매체의 종류와 상관없이 동작하게 해주는 추상 계층이다. 파일시스템의 operation 들이 함수 포인터를 이용하여 VFS 의 operation 에 오버로딩 되는 방식으로 각 파일시스템마다 operation 들이 구현된다. VFS 는 superblock, inode, file, dentry 의 네 가지 data structure 로 이루어진다. Superblock 은 마운트 된 파일 시스템의 정보로, 파일시스템마다 하나씩 존재한다. Inode 는 특정 파일이나 디렉토리에 대한 권한, 타임스탬프, block map 등의 정보이다. File 은 열린 파일에 대한 정보이고, 현재 위치를 file pointer 로 가지고 있다. Dentry 는 특정 디렉토리의 항목에 대한 정보로, 이름과 inode number 를 포함한다.

- Ext4

	Ext2	Ext3	Ext4
Introduced	in 1993	in 2001	in 2006
Max file size	16GB~2TB	16GB~2TB	16GB~16TB
Max file system size	2TB~32TB	2TB~32TB	1EB
Feature	no Journaling	Journaling	Extents Multiblock allocation Delayed allocation

Ext4 (extended file system 4)는 리눅스에서 쓰이는 저널링 파일 시스템(journaling file system) 중 하나로, ext3에서 성능, 확장성 및 신뢰성을 확장시킨 버전이다. ext4는 최대 1EB(1000PB)의 볼륨과 16TB의 파일을 지원하며, ext3와의 쌍방 호환성을 제공한다. 또한 서브디렉토리 제한도 32000 깊이에서 64000으로 확장되었는데, 디렉토리 인덱싱이 해시된 B-tree 형태로 최적화되었기 때문에 제한이 확장되었음에도 불구하고 빠른 조회가 가능하다.

저널링은 저널을 통해 파일 시스템의 변경 사항을 기록하는 프로세스이다. 그런 다음 로그에 기록된 변경 사항에 따라 물리적 스토리지에 실제 변경 사항이 적용된다. 이 방법을 사용하면 좀 더 안정적으로 변경 사항을 구현할 수 있으며 작업 중에 시스템 오류 또는 전원 문제가 발생하더라도 일관성을 유지할 수 있다.



Ext4는 익스텐트 (Extent), 다중 블록 할당 (Multiblock allocation), 지연 할당 (Delayed allocation) 등을 이용하여 입출력 성능을 향상시켰다. 기존의 파일 시스템은 각각의 블록들의 매핑을 유지하기 위해 indirect block을 사용했는데, 이는 큰 파일에 대해서 비효율적이었다. 이를 개선하기 위해 ext4에서는 어느 정도의 연속된 공간만 초기에 할당하고, 추후 그 양이 충분하지 않을 때 또 다른 연속된 공간을 익스텐트 단위로 할당한다.

기존 파일 시스템은 단일 블록만을 다루기 때문에 매 호출마다 단일 블록을 할당했는데, ext4는 이러한 오버헤드를 피하기 위해 한 번의 호출로 많은 블록을 할당할 수 있는 다중 블록 할당을 사용한다. 또한 파일이 캐시에 유지되고 있는 동안에는 그 파일이 실제로 디스크에 쓰여질 때까지 블록 할당을 지연시키는 지연 할당을 사용한다.

- Log structured File System (LFS)

Log structured file system(LFS)는 small write 가 여러 번 발생할 때 seek 으로 인해 성능이 저하되는 것을 막기 위해 disk write 를 모아서 연속된 log 로 처리하는 시스템이다. 마지막에 write 한 블록 뒤로 append 하는 방식이기 때문에 crash 가 발생할 경우 consistency 를 위해 디스크 전체를 스캔해야 하는 다른 파일시스템과는 달리 파일의 제일 뒤에 있는 정보만 체크하면 된다는 장점이 있다.

- Nilfs2

Nilfs2 는 일본의 NTT 에서 개발한 Log structured file system (LFS)이다. 2009 년에 Nilfs2 파일 시스템이 주요 리눅스 커널에 포함되었다. 주요 특징은 연속적인 스냅샷을 할 수 있다는 것이다. 사용 중에 스냅샷을 생성할 수 있기 때문에 백업을 만들어내기에 편리하다.

- Loadable Kernel Module (LKM)

Loadable Kernel Module은 시스템 운영 중에 동적으로 커널에 load와 unload가 가능한 모듈을 의미한다. 이를 통해 Monolithic-kernel에서도 Micro-kernel의 장점을 취할 수 있게 되는데, 예를 들어 memory management나 process management, Virtual File System layer 등 커널에서 절대 빠져서는 안 되는 핵심적인 부분들은 monolithic component로서 존재하고, 그 외 device driver나 file system 등은 loadable component로 존재하는 분리된 구조를 갖게 된다.

이러한 Loadable Kernel Module은 커널 기능을 쉽게 확장할 수 있게 해주고, 커널의 주소 공간 안에서 수행되기 때문에 성능의 저하가 일어나지 않는다. 이러한 장점도 있지만, LKM이 커널 영역 안에서 동작하기 때문에 모듈이 잘못 작성되었을 경우에는 시스템 전체의 동작이 중지될 수도 있다는 단점 또한 갖고 있다.

LKM을 작성하기 위해서는 Makefile과 C 소스 코드를 작성하면 된다. 이 소스 코드를 컴파일하면 생기는 *.ko 파일이 LKM 파일이 된다. LKM을 작성할 때는 커널 소스가 필요한데, 해당 LKM을 올리려는 커널과 버전이 같아야만 동작이 가능하다. 버전이 같지 않으면 모듈을 커널에 올릴 수 없게 된다. Makefile에 커널 소스의 경로를 지정해 주면, make를 수행했을 때 해당 커널 소스를 참고하여 컴파일이 이루어진다.

작성한 모듈을 커널에 올릴 때는 insmod 명령어를, 커널에서 내릴 때는 rmmod 명령어를 사용한다. 현재 커널에 올려져 있는 모듈의 목록을 확인할 때는 lsmod 명령어를 사용하고, modinfo 명령어로 version, license, description, author 등 모듈의 정보를 확인할 수 있다.

- Proc File System

Proc file system은 메모리에만 존재하는 파일 시스템으로, in-kernel data structure들을 사용자가 읽거나 제어할 수 있도록 인터페이스를 제공해 준다. 사용자는 proc file system을 통해 시스템에 대한 정보를 얻거나, 런타임 중에 커널 파라미터를 변경할 수 있다.

커널 코드를 개발하며 필요한 정보를 출력하기 위해서는 printk()를 사용한 후, dmesg를 통해 커널 내의 로그 버퍼를 출력하는 방식을 사용할 수 있다. 그러나 이러한 방법은 버퍼의 크기가 제한되어 있고, 콘솔로 바로 출력되기 때문에 데이터를 가공하기 어렵다는 한계가 있다. 이러한 한계를 극복하기 위하여 proc file system을 사용할 수 있다. proc 내부에 파일을 만들어 read(), write() 시스템 콜을 활용하면 더욱 편리하게 조작이 가능하다.

proc 내부의 디렉토리와 파일을 새로 만들기 위해서는 해당하는 모듈 코드를 작성하면 된다. proc 파일을 만들 때는 proc_create를 사용하는데, 이때 해당 파일을 open, read, write할 때 사용할 operation들을 인수로 함께 넣어 주어야 한다.

소스 코드 설명

- blk-core.c

```
#define Q_SIZE 1024
//순환큐 크기

struct elem{
    unsigned long long block_n; //블럭 넘버
    long time; //write 발생 시간
    char* fs_name; //fs 종류
};

struct elem buffer[Q_SIZE]; //순환큐 선언
int q_index = 0; //순환큐의 현재 인덱스를 나타냄
```

디스크에 쓰기가 발생한 순간 해당 block number 와 발생 시간, 파일시스템의 종류를 저장하기 위한 큐를 만들어야 한다. elem 은 write 이 한 번 일어났을 때 저장할 정보를 모두 포함하는 구조체이다. elem 의 배열로 'buffer'라는 이름의 큐를 만들었고, 큐의 크기는 Q_SIZE 라는 상수로 선언하였다. 큐에 elem 을 넣기 위해서는 어느 위치에 넣어야 할지를 알아야 하기 때문에 현 위치를 표시할 q_index 라는 변수를 만들고 0 으로 초기화 하였다.

```
void elem_in(struct elem item) {//큐에 원소를 추가하는 함수
        buffer[q_index].block_n = item.block_n; //블럭넘버 저장
        buffer[q_index].time = item.time; //발생시간 저장
        buffer[q_index].fs_name = item.fs_name; //파일시스템 이름 저장
        q_index = (q_index + 1) % Q_SIZE; //다음 인덱스 지정. 순환큐이기 때문에 큐가 다 찼을 경우 처음부터 다시 저장하기 위해 모듈러 연산 사용
}
```

큐에 원소를 추가하는 elem_in()함수를 위와 같이 만들었다. 순환 큐이기 때문에 큐가 다 찼을 경우 처음부터 다시 저장하기 위해서 Q_SIZE modulo 연산을 이용하였다.

```
if (rw & WRITE) {
    count_vm_events(PGPGOUT, count);
    if (bio && bio->bi_bdev && bio->bi_bdev->bd_super && bio->bi_bdev->bd_super->s_type) {//null인 경우에 대한 exception 처리
        char *fs_name;
        fs_name = bio->bi_bdev->bd_super->s_type->name;//file system 이름을 저장해준다
        struct elem item;//큐에 들어갈 새로운 원소 생성
        item.block_n = bio->bi_iter.bi_sector; //블럭 넘버 저장
        item.time = current_fs_time(bio->bi_bdev->bd_super).tv_sec; //write 발생 시간 저장
        item.fs_name = fs_name; //file system 이름 저장
        elem_in(item);//큐에 원소를 넣는다
    }

} else {
```

submit_bio() 함수에서 write 발생시 실행되는 부분이다. 가장 먼저 if 문으로 포인터가 null 값을 갖는 경우에 대한 예외처리를 해준다. bio 는 I/O 의 단위를 나타내는 구조체이고, bi_bdev 는 block device 에 대한 정보를 갖는 멤버 변수이다. bd_super 는 superblock 변수로, 안에 s_type 이라는 file system 의 종류에 대한 정보를 저장하고 있다. 여기에서 파일시스템의 이름에 해당하는 name 을 가져온다. Block number 는 bi_sector 에 접근하여 가져올 수 있다(linux/blk_types.h 에 device address in 512 byte sectors 라는 주석이 달려있다). 시간 정보는 superblock 을 매개변수로 받아서 파일시스템 시간을 리턴하는 current_fs_time()함수를 사용하였다.

```
//lkm 모듈에서 참조하기 위해 순환큐와 큐의 현재 인덱스 export
EXPORT_SYMBOL(buffer);
EXPORT_SYMBOL(q_index);
```

lkm 소스 파일에서 큐와 그 인덱스에 접근하기 위해 export 해 주었다.

```
if(!strcmp(fs_name, "nilfs2")){
    struct elem item;//큐에 들어갈 새로운 원소 생성
    item.block_n = bio->bi_iter.bi_sector; //블럭 넘버 저장
            item.time = current_fs_time(bio->bi_bdev->bd_super).tv_sec; //write 발생 시간 저장
    item.fs_name = fs_name; //file system 이름 저장
    elem_in(item);//큐에 원소를 넣는다
}
```

단, nilfs2 의 write 측정을 진행할 때는 파일시스템의 이름이 nilfs2 일 경우에만 큐에 정보를 저장하도록 if 문을 추가하였다.

- segbuf.c

```
//block device의 bd_super가 Nilfs2의 superblock을 참조하도록 수정
if(bio && bio->bi_bdev){
    bio->bi_bdev->bd_super = segbuf->sb_super;
}
bio->bi_end_io = nilfs_end_bio_write;
bio->bi_private = segbuf;
submit_bio(mode, bio);
```

nilfs_segbuf_submit_bio()내에서 submit_bio()를 호출하기 전에 block device 의 superblock 이 nilfs2 의 superblock 을 참조하도록 if 문을 하나를 추가하였다.

- lkm.c

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

#define Q_SIZE 1024 //순환큐 크기
char proc_buffer[Q_SIZE][33]; //순환큐에서 정보를 읽어와 문자열 형태로 담아 둘 buffer

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

struct elem {
    unsigned long long block_n; //블럭 넘버
    long time; //write 발생 시간
    char* fs_name; //fs 종류
};
extern struct elem buffer[Q_SIZE]; //blk-core.c에서 선언한 순환큐
extern int q_index; //순환큐의 현재 인덱스
```

proc 파일 내부에서 사용하는 버퍼의 역할을 할 proc_buffer와 proc_dir_entry 변수들을 선언한 후, 커널의 순환 큐에서 사용하는 구조체를 정의하고 커널 내부 순환 큐와 인덱스를 extern으로 선언하였다.

```c
static int my_open(struct inode *inode, struct file *file) {
    printk(KERN_INFO "Module Open\n"); //파일 오픈 시 간단한 메세지 출력
    return 0;
}
```

proc 파일 오픈 시 간단한 메시지를 출력하도록 my_open을 정의하였다.

```c
static ssize_t my_write(struct file *file, const char __user * user_buffer, size_t count, loff_t *ppos) {
    printk(KERN_INFO "Module Write\n");

    int p_index = q_index; //커널에서 다음에 입력할 인덱스의 값을 받아옴 = 현재 큐에 들어있는 값들 중 가장 오래된 정보를 가리킴
    int i;
    for (i = 0; i < Q_SIZE; i++) { //p_index에서 시작해서 Q_SIZE만큼 반복(한 바퀴 순회)하면서 가장 오래된 값부터 문자열로 저장
        sprintf(proc_buffer[i], "(%10lld, %10ld, %6s)\n", buffer[p_index].block_n, buffer[p_index].time, buffer[p_index].fs_name);
        p_index = (p_index + 1) % Q_SIZE; //다음으로 가져올 인덱스 지정. 순환큐임을 고려하여 모듈러 연산
    }
    return count;
}
```

커널 내부 순환 큐에 저장된 값들 중 가장 오래된 값을 시작으로 한 바퀴 돌며, 큐에 들어 있는 값들을 proc_buffer에 저장하도록 my_write를 정의하였다. proc_buffer에 저장할 때는 (block_n, time, fs_name)의 형식의 문자열로 저장하기 위해 sprintf를 사용하였다.

```c
static ssize_t my_read(struct file *file, char __user * user_buffer, size_t count, loff_t *ppos) {
    printk(KERN_INFO "Module Read\n");

    unsigned long proc_count = sizeof(proc_buffer); //user_buffer에 넘겨줄 바이트 수
    copy_to_user(user_buffer, proc_buffer, proc_count); //proc_buffer에 저장된 값을 user_buffer에 넘겨 줌

    *ppos += proc_count; //읽은 바이트수만큼 ppos 이동
    if (*ppos > proc_count){
        return 0; //더이상 읽을 것이 없음
    } else {
        return proc_count; //읽은 바이트수 리턴
    }
}
```

copy_to_user를 사용하여 proc_buffer에 들어 있는 값을 user_buffer로 넘겨주는 my_read를 정의하였다. 이때 읽은 바이트 수만큼 ppos를 이동시키고, 더이상 읽을 값이 없는 경우 0을 return하고, 그렇지 않은 경우 읽은 바이트 수를 return하도록 작성하였다.

```c
static const struct file_operations myproc_fops = { //file_operations 선언
    .owner = THIS_MODULE,
    .open = my_open,
    .write = my_write,
    .read = my_read,
};
```

위에서 정의해준 my_open, my_write, my_read를 myproc_fops 구조체로 선언하였다.

```c
static int __init simple_init(void) {
    printk(KERN_INFO "Module Init\n");

    proc_dir = proc_mkdir(PROC_DIRNAME, NULL); //proc directory 생성
    proc_file = proc_create(PROC_FILENAME, 0600, proc_dir, &myproc_fops); //proc file 생성
    return 0;
}
```

모듈이 올라갈 때 proc directory와 proc file을 생성하도록 simple_init을 정의하였다.

```c
static void __exit simple_exit(void) {
    printk(KERN_INFO "Module Exit\n");

    proc_remove(proc_file); //proc file 제거
    proc_remove(proc_dir); //proc directory 제거
    return;
}
```

모듈이 내려갈 때 proc file과 proc directory를 제거하도록 simple_exit을 정의하였다.

```
module_init(simple_init);
module_exit(simple_exit);

MODULE_DESCRIPTION("Proc Write & Read Module");
MODULE_LICENSE("GPL");
MODULE_VERSION("NEW");
```

모듈이 커널에 올라갈 때와 내려갈 때 실행할 함수를 지정해 주고, MODULE_LICENSE를 GPL로 지정하였다.

- Makefile

```
obj-m += lkm.o

KDIR = /usr/src/linux-4.4

all:
    $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules
clean:
    $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) clean
```

lkm.c 소스 코드로부터 lkm.ko 모듈을 만들기 위하여 **obj-m += lkm.o**를 작성하였고, KDIR에 linux-4.4 커널의 소스 코드가 들어 있는 경로를 주었다.

실행 방법에 대한 간략한 설명

1. /usr/src/linux-4.4 에서 blk-core.c 를 수정한다.

2. sudo make 와 sudo make install 을 실행한 후 재부팅한다.

3. LKM 폴더에 작성 파일(Makefile, c 소스파일)을 추가한 후 make 로 컴파일한다.

4. 커널에 모듈을 올린다. (sudo insmod [.ko 파일명]. 이번 과제에서는 소스파일명이 lkm.c 이므로 sudo insmod lkm.ko 가 된다)

5. iozone 디렉토리에서 iozone 을 실행한다. (./iozone -i 0 -f [임시파일위치])

6. sudo sh -c "echo [data] > /proc/[directory name]/[file name] "을 실행한다. (LKM 작성파일 lkm.c 에서 directory name, file name 모두 myproc 으로 지정함)

7. sudo cat /proc/myproc/myproc > result.txt 을 실행해서 데이터를 읽고, 그 결과를 result.txt 에 저장한다.

8. nilfs2 의 write 측정을 위해 blk-core.c 파일과 segbuf.c 파일을 수정한다. blk-core.c 에는 nilfs2 의 write 만을 측정하기 위한 if 문을, segbuf.c 에는 nilfs2 의 superblock 을 지정하는 코드를 추가한다. 그 후 재부팅한다.

9. nilfs2 디스크를 만들고 디렉토리에 mount 한다.

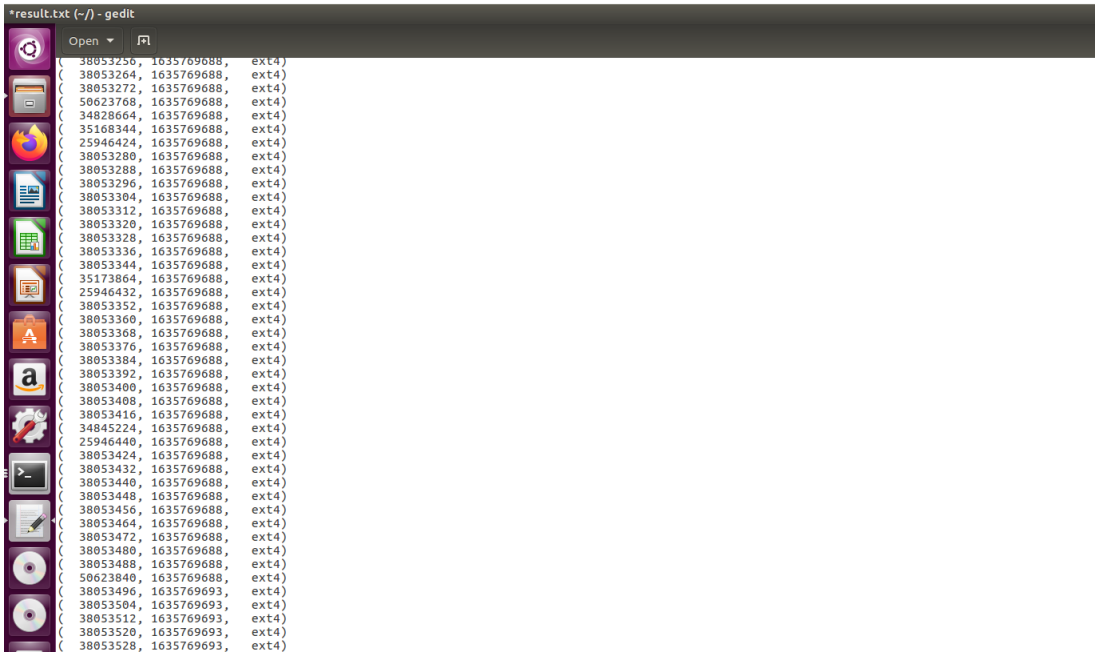
10. 앞에서 나온 5~7 을 다시 실행한다. 이 때 iozone 의 임시파일 위치는 nilfs2 를 마운트한 디렉토리이다.

실행 결과 캡처 화면

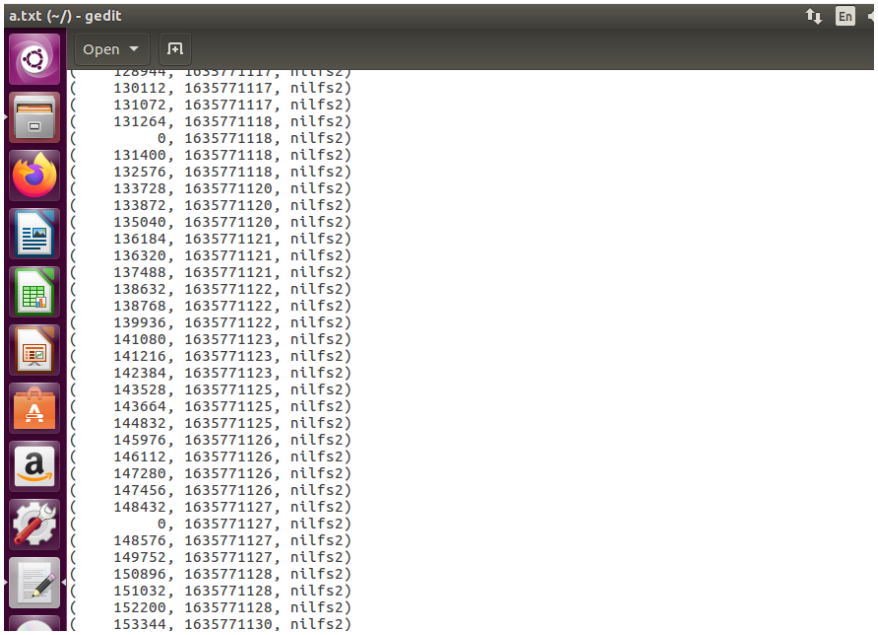
-실행이 끝난 후 dmesg

```
[27957.419625] Module Init
[28015.176883] Module Open
[28015.176905] Module Write
[28028.194354] Module Open
[28028.194405] Module Read
[28028.194859] Module Read
[28136.857831] Module Exit
```

-sudo cat /proc/myproc/myproc >/result.txt 실행 시 결과 화면 (ext4)

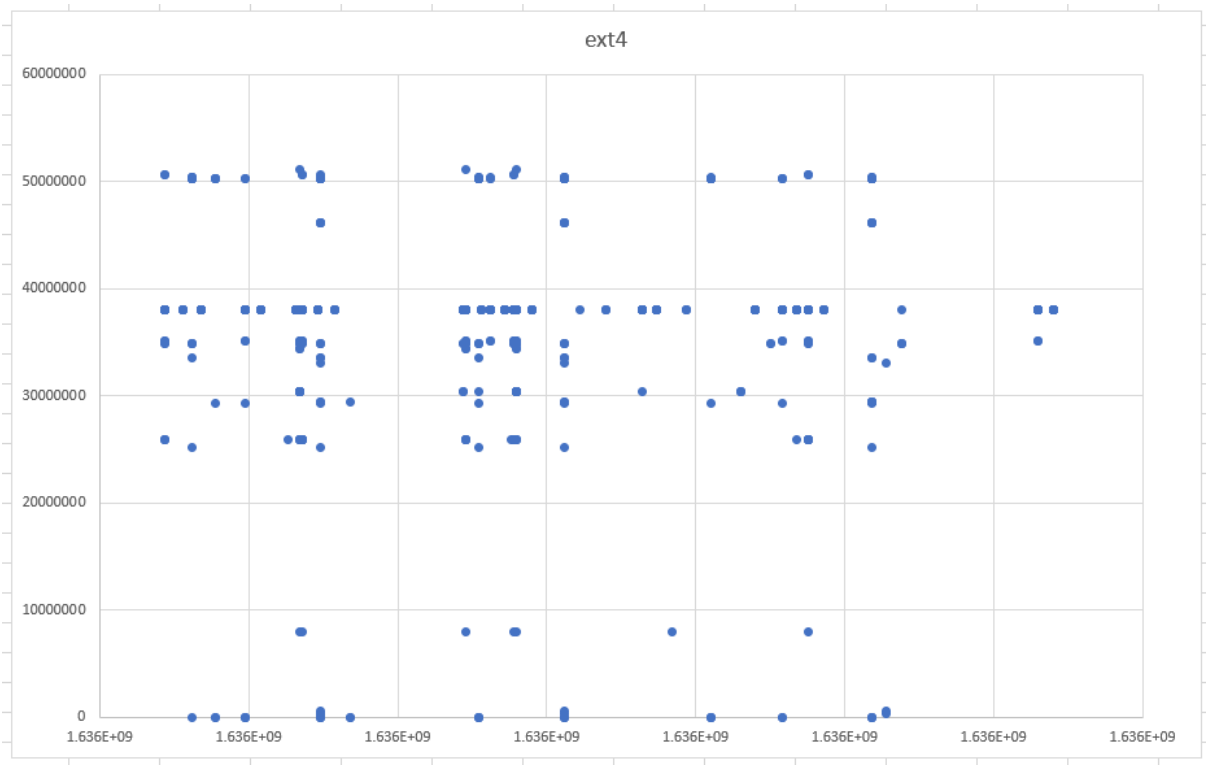


--sudo cat /proc/myproc/myproc > /a.txt 실행 시 결과 화면 (nilfs2)



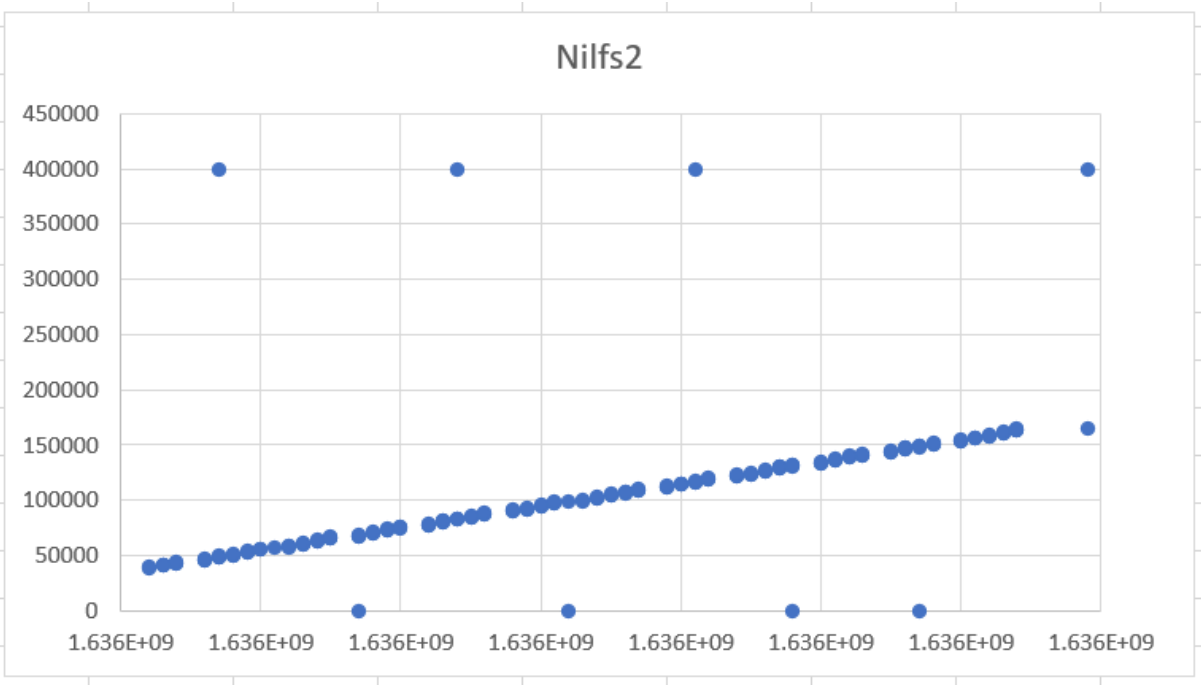
결과 설명

- Ext4의 쓰기 패턴



그래프를 보면, ext4에서 write를 수행할 때 연속적인 블록에 접근하지 않고 불연속적으로 떨어져 있는 블록에 접근하는 모습을 확인할 수 있다. Ext4는 FFS(Fast File System) 방법을 채택했기 때문에 한 실린더 그룹에 superblock, inode, file들을 배치하고, 같은 디렉토리에 속하는 파일과 inode를 같은 실린더 그룹에 할당한다. 여기서 inode와 데이터 블록의 위치는 분리되어 있다. 따라서 같은 실린더 그룹이라고 하더라도 inode와 data block에 write하기 위해서는 각각의 블록에 따로 접근해야 하므로 non-sequential한 write를 수행하게 되어, 그래프와 같이 불연속적인 형태를 보인다.

-Nilfs2의 쓰기 패턴



그래프이 x 축은 시간, y 축은 block number 이다. Nilfs2 는 배경지식 설명에서도 언급했듯이 sequential 하게 append 하는 LFS 의 일종이기 때문에 시간이 지남에 따라 block number 가 증가한다. 다만 중간에 주기적으로 0 번, 399992 번 block 에 write 을 하는데, 이는 스냅샷을 하기 위해 체크포인트(복구 포인트)를 작성하는 것으로 추정된다. 아래는 한국데이터산업진흥원에 올라와있는 nilfs2 의 체크 포인트에 대한 설명이다.

> NILFS 는 로그 형태로 구조화되어 있기 때문에 새 데이터는 로그의 헤드에 쓰여지며 기존 데이터는 가비지 컬렉션되지 않는 한 계속 존재한다. 기존 데이터가 계속 존재하기 때문에 원하는 시점으로 돌아가서 해당 파일 시스템의 에포크(epoch)를 조사할 수 있다. 이러한 에포크는 NiLFS(2)에서 체크포인트라고 불리며 파일 시스템의 핵심 부분이 된다. NiLFS(2)에서는 파일 시스템에 변화가 생길 때마다 이러한 체크포인트가 작성되지만 사용자가 강제로 체크포인트를 작성할 수도 있다. 체크포인트(복구 포인트)는 스냅샷으로 변경될 수 있을 뿐만 아니라 볼 수도 있다.[1]

[1] 한국데이터산업진흥원, <차세대 Linux 파일 시스템인 NiLFS(2)와 exofs 로그와 오브젝트를 사용하는 고급 Linux 파일 시스템>
https://www.kdata.or.kr/info/info_04_view.html?field=&keyword=0&type=techreport&page=107&dbnum=148449&mode=detail&type=techreport

- Nilfs2와 Ext4의 쓰기 동작의 차이

Nilfs2는 block number가 연속적인 반면 ext4는 불연속적이다. Nilfs2는 여러 개의 small write를 모아뒀다가 한 번에 sequential한 single large write을 하고, ext4는 write가 발생할 때마다 non-sequential write를 한다는 것을 확인할 수 있다.

- Ext4의 쓰기 패턴을 F2FS와 비교

F2FS(Flash-Friendly File System)는 플래시 특성에 맞는 플래시 파일 시스템으로, log-structured file system에 기초하여 설계되었다. 플래시 메모리는 읽기 속도가 빠르지만 쓰기 속도가 느리고, 데이터의 덮어쓰기가 불가능하며, 블록을 지울 수 있는 횟수가 유한하다는 특징을 갖고 있다. 이러한 특성에 맞게 F2FS는 6개의 섹션을 활용하여 데이터를 hot, warm, cold로 나눠서 관리하며 garbage collection을 수월하게 하고, 이 6개의 섹션이 각각 write를 수집하고, large write를 수행한다. 이와 비교하여 ext4는 데이터를 hot, warm, cold로 나눠서 관리하지 않으며, write 명령이 들어올 때마다 non-sequential한 write를 수행하는 쓰기 패턴을 보인다.

과제 수행 시 어려웠던 부분과 해결 방법

- sudo echo a > myproc 수행 시 Permission denied 오류

LKM을 통해 만들어진 myproc 파일에 write를 수행하기 위하여 sudo echo a > myproc을 실행하였을 때, Permission denied 오류가 발생하여 제대로 실행되지 않았다. 이를 해결하기 위하여 조사를 해 보니, shell이 echo 명령에는 sudo를 적용하지만 redirection에는 원래 사용자 권한을 그대로 남겨두기 때문에 해당 문제가 발생했다는 것을 알 수 있었다. 이를 참고하여 sudo echo a > myproc 대신 sudo sh -c "echo a > myproc"을 수행하였더니 문제가 해결되었다.

- cat 실행 시 read 무한 반복

proc file system을 위한 LKM을 작성하던 과정에서 my_read의 return값을 단순히 proc_count로 설정하였더니, cat 실행 시 my_read 함수를 계속해서 호출하는 문제가 발생하였다. 이를 해결하기 위하여 read()에서 return 값이 갖는 의미에 대하여 다시 찾아본 후, return 값이 0일 때 end of file을 의미한다는 사실을 이용하여 함수를 수정하였더니 더 이상 문제가 발생하지 않고 정상적으로 작동하였다.

- init 함수 내 무한루프 구현 시 문제

LKM의 simple_init 함수 내에 1초마다 순환 큐에 있는 정보를 읽어 오는 반복문을 작성하였더니, insmod가 실행된 이후 rmmod를 제대로 수행할 수 없는 문제가 발생하였다. 이를 해결하기 위하여 simple_init 함수 내에 반복문을 두지 않고, my_write 함수에서 순환 큐의 값을 읽어 오도록 수정하였다.