COSE322-00

SYSTEM PROGRAMMING

1st Term Project: Log Structured File System & Ext4 File System Profiling

2016.11.02

GROUP 18

2010100086 김학영, 2012130888 김인호

Prof. 유혁

Korea University

TABLE OF CONTENTS

1. 시작하기

역할 분담

커널 코드는 주로 김인호가 담당하고, 커널 모듈 코드는 김학영이 주로 담당하였다. 서로 모르거나 부족한 부분은 상호보완해가면서 진행하였다. 커널 코드 부분 중 blk-core.c 파일에서 디스크 섹터 번호, 접근 시간, 파일 시스템 이름을 참조하여 출력하는 부분을 김인호가 구조화하였고, 예외처리와 컴파일 오류에 대한 디버깅을 김학영이 수행하였다. 접근 시간 단위를 microsecond단위로 수정하고, super block에 접근할 때의 NULL Pointer Exception 등 커널 컴파일 과정에서 발생하는 crash 문제들을 김학영이 주로 처리하였다. LKM인 hw1.c는 김인호가 커널 전역 변수를 통해 커널에서 저장한 정보를 모듈로 받아오고 기본 함수 구조 및 proc file operation에 대해 정의하여 결과 측정을 수행할 수 있는 기반을 다졌다. 이에 이어 김학영이 결과를 csv 파일로 출력하여 마이크로소프트 엑셀 프로그램과 호환성을 높이는 방안을 고안하여 추가하였다. 그리고 유저 영역에서 proc 파일 시스템을 통해 사용자로부터 파일시스템명을 입력받아서 원하는 결과만을 출력하도록 필터링하는 과정을 김학영이 구현하였다.

배경지식

리눅스 환경에서 ext4와 같은 일반적인 파일 시스템은 파일을 저장할 때 생성 시간, 수정 시간, 접근 권한 등 다양한 정보를 포함하는 메타데이터를 inode라는 것에 저장한다. 그리고 이 inode는 실제 파일의 데이터가 저장된 블록들의 위치 정보를 가지고 있다. 이 inode는 dentry라는 또 다른 구조체에 포함된다. inode를 식별하기 위한 고유 번호(inode number)와 파일명이 함께 묶여서 dentry라는 구조체가 되고, 이를 통해 사용자는 파일에 접근할 수 있게 된다. 리눅스 터미널에서 ls -i를 입력하면 inode의 번호와 파일 이름이 맵핑되어 있는 것을 볼 수 있다.

이렇게 파일의 메타데이터를 가지고 있는 inode들은 물리 디스크 상에 흩어져 저장되기 때문에 어느 위치에 저장되어 있는지 tracking을 할 필요가 있다. 따라서 super block이라는 구조체가 inode들의 위치 정보를 가지고 있게 된다. 따라서 ext4에서 특정 파일에 write를 수행하면, 물리 디스크 상에서 서로 떨어져 있는 여러 블록에 분산되어 write 작업이 발생할 것이다. 이렇게 흩어져 있는 데이터들은 super block을 통해 해당 파일의 inode의 위치를 알아내고, 그 inode를 참조하여 실제 데이터들이 저장되어 있는 곳을 찾는 과정을 통해서 읽고 수정할 수 있다.

그런데, nilfs와 같은 Log Structured 파일 시스템은 다른 양상으로 이루어져 있다. 하나의 파일에 write를 할 때 여러 블록들이 떨어진 곳에 위치해 있어서 디스크 접근 시간이 오래 걸린다는 문제점을 보완하기 위해 write 작업을 연속된 공간에 순차적으로 진행한다. 즉, 떨어져 있는 공간이 아니라 연속된 디스크 블록에 실제 데이터들을 쓰고, 그에 대한 메타데이터도 역시 따로 특정한 곳에 저장을 하는 것이 아니라 실제 데이터와 함께 연속적으로 저장을 한다. 새로운 파일 생성하

는 것뿐만 아니라, 기존의 파일을 수정할 때도 ext4처럼 inode를 통해 실제 데이터 블록들의 위치 정보를 참조하여 직접 수정하는 것이 아니라, 그냥 마지막으로 쓰기를 마친 지점부터 순차적으로 새로 쓰기를 진행한다. 즉, 계속 append 작업만 수행하는 것이다.

그런데, 이는 이전 데이터를 계속 디스크 상에 남겨 놓는다는 점에서 필연적으로 디스크 공간을 낭비할 수 밖에 없다. 따라서 지속적으로 garbage collection 과정을 통해 디스크 공간의 여유분을 확보해주어야 한다. 그리고 또한, 파일 읽기를 수행할 때의 문제점이 있다. 쓰기는 순차적으로 진행하기 때문에 매우 간편하고 효율적이지만, 파일 읽기의 경우에는 무차별적인 append 작업에 의해 어디에 파일이 저장되어 있는지를 알 수 없기 때문이다. ext4의 super block 처럼 inode들의 위치 정보를 저장해둘 것이 필요하다. 이러한 문제를 해결하기 위해, inode map이라는 것을 메모리에 caching해서 수많은 inode의 copy들 중에 어떤 것이 가장 최근의 유효한 inode인지 그 정보들을 저장해둔다.

2. 개발 과정

Kernel Source Code - 기본 뼈대 구성

ext4 또는 nilfs 모두 write 작업이 수행될 때 blk-core.c 커널 소스코드에 정의되어 있는 submit_bio라는 함수를 거치게 된다. 이 함수에서 쓰고자하는 디스크 섹터의 정보를 얻을 수 있기 때문에 이 함수를 중점적으로 수정하였다.

우선, LKM 쪽에서 커널에서 얻은 정보를 사용하기 위해서 커널 소스 내부에 전역변수 형태로 정의를 해두기로 하였다. 디스크의 섹터 정보를 저장하기 위한 큐, write가 수행된 시간을 저장하기 위한 큐, write가 수행되고 있는 파일 시스템의 이름을 저장하기 위한 큐로 총 3개의 환형 큐를 정의하였다. 처음에는 큐의 크기를 1024로 설정했지만, 다소 적다고 판단하여 2000개로 늘렸다.

```
54  // ----- added ----- by Inho //
55  unsigned long long hw1_buffer[2000];
56  EXPORT_SYMBOL(hw1_buffer);
57  int hw1_index = 0;
58  EXPORT_SYMBOL(hw1_index);
59  long long int hw1_time[2000];
60  EXPORT_SYMBOL(hw1_time);
61
62  // ----- addedd ----- by Hakyoung //
63  const char* hw1_file_system_type[2000];
64  EXPORT_SYMBOL(hw1_file_system_type);
65
66  struct timeval mytime;
67  // ----- added -----//
```

환형 큐 자료구조의 경우 별다른 자료구조를 만들지 않고 간단하게 배열을 사용하였다. 필요한 작업들을 완료한 뒤에 Modular 연산을 통해 인덱스를 조정하여 환형 큐 형태가 되도록 하였다.

```
hw1_index = (hw1_index + 1) % 2000;
```

(1) 디스크 섹터 정보 출력

디스크 섹터 정보는 비교적 쉽게 얻을 수 있었다. submit_bio 함수에서 bio 구조체의 멤버로 존재하는 bi_sector를 통해 참조할 수 있었다. include/linux/blk-types.h를 직접 찾아보니 bio 구조체 내부에 bi_iter라는 멤버를 통해 bi_sector에 접근을 할 수 있다는 것을 알아냈다. 따라서 이를 unsigned long long형으로 캐스팅하여 큐에 저장하였다. 그런데, 몇 번의 테스트 과정을 거쳐보니 submit_bio 함수에서 virtual memory에 0에 접근한다며 kernel crash가 발생했다. 따라서 bi_sector에 접근할 때 우선적으로 NULL인지 체크해주는 조건문을 추가하여 문제를 해결하였다.

```
// ---- added ---- by Inho (16. 10. 26) // // ---- moved ---- by Hakyoung (16. 10. 28)
//
if(bio->bi_iter.bi_sector!=NULL){    //should check if bi_sector is NULL. Otherwise,
meaningless values will be included.
    hw1_buffer[hw1_index] = (unsigned long long) bio->bi_iter.bi_sector;
```

(2) 섹터 접근 시간 출력

섹터 접근 시간을 출력하기 위해서 현재 시간을 구할 수 있는 여러 함수에 대해 조사해봤는데, 그 중 gettimeofday와 clock_gettime이라는 두 함수를 찾았다. gettimeofday의 경우 앞으로 유닉스 표준에서 제외될 것이라는 정보를 얻어서 되도록이면 clock_gettime을 사용하고자 하였다. 그런데, 이 함수에서 사용되는 clock_gettime은 nanosecond 단위로 측정을 하기 때문에 너무 작다고 판단되어 결국 microsecond 단위로 측정을 해주는 gettimeofday 함수를 사용하기로 하였다. 커널이 아닌 유저 영역에서의 c 파일로 테스트 과정을 거쳤는데, 문제가 발생했다. 출력되는 시간 값들이 어째서인지 꾸준히 증가하는 것이 아니라 자꾸 작은 값으로 되돌아가는 현상이 관찰되었다. 문제는 gettimeofday에 사용되는 timeval이라는 구조체를 잘못 이해한 데에 있었다. 구조체의 멤버로 tv_usec이라는 것이 microsecond 단위로 현재 시간을 측정하는 것인데, 이는 second 단위 아래의 millisecond와 microsecond만 출력을 하기 때문에 값이 맴돌았던 것이다. 따라서 같은 구조체 내에 현재 시간을 second 단위로 측정해주는 tv_sec이라는 변수에 1,000,000을 곱한 다음 tv_usec에 더함으로써 문제를 해결하였다.

그런데, 이 함수를 커널 영역에서 사용하고자 하니까 implicit declaration of function이라는에러가 발생하여 컴파일이 정상적으로 되지 않았다. 이는 gettimeofday라는 함수를 정의하는 sys/time.h라는 헤더파일이 include되지 않았기 때문에 발생한 문제로 판명되었다. 따라서 blk-core.c 파일에 이 헤더파일을 include하고자 하였으나, 커널 소스 디렉토리 내에 존재하는 헤더파일이 아니어서

결국 실패하였다. 따라서 대안으로 time.h 헤더파일에 정의되어 있는 CURRENT_TIME 이라는 매크로 상수를 사용하였다. 이는 current_kernel_time()이라는 함수를 가리키며, 이 함수는 커널의 현재 시간을 nanosecond단위까지 알려주는 timespec 구조체를 반환한다. 섹터 접근 속도는 1초에 수천, 수만번 발생할 수 있기 때문에 현재 시간을 나노초까지 불러오지 못하면 결과 데이터에 같은 시간 값을 가진 레코드가 생긴다. 따라서 이 둘을 모두 고려하기 위해 tv_sec에 1,000,000,000을 곱한 값에 tv_nsec을 더하여 나노초 단위로 현재 시간을 나타내는 long long 타입의 값을 hw1_time 배열에 추가했다. 그러나 예상과 다르게 실제 출력하는 시간값이 중복되는 레코드가 상당히 많이 발생했다.

그런데, 결국 microsecond 단위로 측정을 해주는 do_gettimeofday라는 함수를 발견하여 microsecond단위로 디스크 섹터의 접근시간을 얻어낼 수 있었다. 따라서 이를 큐에 저장하고, LKM 쪽에서 참조할 수 있도록 하였다.

```
do_gettimeofday(&mytime);
hw1_time[hw1_index] = (unsigned long long)(mytime.tv_sec) * 1000000 + (unsigned long
    long)(mytime.tv_usec);
```

(3) 파일 시스템 이름 참조

파일 시스템 이름을 참조하기 위해서 lxr의 여러 페이지를 참조하였다. 우선 bio 구조체의 멤버 중 block_device라는 구조체를 include/linux/fs.h 헤더파일에서 찾을 수 있었다. 이 구조체가 멤버로 super_block이라는 구조체 형태로 슈퍼 블록의 정보를 가지고 있었다. 같은 헤더파일에서 찾아보니 super_block 구조체 내에는 멤버로 file_system_type이라는 구조체가 포함되어 있었다. 이 구조체는 const char* 형태로 name이라는 멤버 변수를 포함하고 있다. 이것이 파일시스템 이름을 나타낼 것이라고 확신하고 bio->bi_bdev->bd_super->s_type->name 순서로 참조를 하였다. 예외처리 없이 곧바로 버퍼에 bio의 block device인 bi_bdev의 슈퍼 블락인 bd_super의 파일시스템 종류를 포함하는 구조체 s_type의 요소 중 name을 넣었더니 컴파일 과정에서 kernel crash가 발생했다. 예외처리의 필요성을 깨닫고, bio->bi_bdev의 NULL 여부, 그 이후 bio->bi_bdev->bd_super의 NULL 여부 등 계속해서 포인터를 참조하는 모든 부분을 한 단계 한 단계 예외처리를 해줬더니 ext4 파일시스템 이름이 정상적으로 출력되었다.

그러나, nilfs2는 그렇지 못했다. fs/nilfs/segbuf.c에서 nilfs2의 슈퍼블락 구조체를 bio가 가리키도록 할 때, 예외처리에서는 먼저 bio->bi_bdev의 NULL 여부, 그 다음 bio->bi_bdev->bd_super의 NULL 여부를 살펴보도록 하였다. 그러나 후자의 예외처리의 경우 해당 block device가 가리키는 super block 주소가 NULL인지 아닌지는 상관없이 그 값에 segbuf->sb_super, 즉 nilfs2의 슈퍼 블락의 주소값을 넣는 것이기 때문에 필요하지 않은 예외처리였다. 과도한 예외처리로 인한 오류였던 것이다. 따라서 해당 예외처리를 없애니 nilfs2 파일시스템 이름 또한 정상적으로 출력되었다.

```
// ----- added ----- by Hakyoung (16. 10. 26) //
if(bio->bi_bdev!=NULL)
    bio->bi_bdev->bd_super = segbuf->sb_super;
// ----- added ----- //
```

포인터를 참조할 때에는 C언어 특성상 예외처리가 필수적이다. 포인터를 참조하는 경우는 크게 세 가지 경우가 있다. blk-core.c의 submit_bio 함수에서 파일시스템과 섹터에 접근할 때, seg_buf.c의 bio의 슈퍼블락이 nilfs의 슈퍼블락 포인터를 참조하도록 할 때, hw1 모듈에서 사용자로부터 echo로 입력받은 char로 접근할 때이다.

LKM – proc을 통한 사용자 디렉토리 접근, 파일 생성 및 쓰기

hw1 모듈이 단순히 printk를 이용하여 로그에 버퍼값을 기록하는 것이 아니라, 곧바로 csv 확장자 파일을 만든다면 데이터 분석이 훨씬 쉬울 것이라고 판단하여 모듈이 사용자의 디렉토리에 접근하여 파일을 생성/수정하는 기능을 추가했다. hw1 모듈에 write가 발생하면 버퍼값에 접근하기 전 set_fs 함수를 통해 커널의 메모리를 해당 사용자 영역 디렉토리의 파일시스템에 맞도록 설정했다. 그리고 file 구조체를 이용하여 /tmp/result.csv 파일을 열어 버퍼의 값들을 입력할 준비를 마친다. IS_ERR 함수는 입력받은 file 구조체가 유효한지, 오류가 없는지를 판단하여 boolean값을 되돌려준다. 이를 이용하여 파일이 열렸을 때에만 다음 과정을 진행하도록 예외처리를 했다.

write 과정에서 큐에 저장된 파일시스템 이름을 유저 영역에서 사용자로부터 echo로 입력받은 문자열과 비교하는 strcmp 함수를 사용하여 필터링한 뒤 버퍼값을 출력했는데, 어떠한 값도 출력되지 않았다. 사용자가 입력한 문자열 주소인 user_buffer를 출력해보니 입력값 뒤에 문자의 끝을 나타내는 값 '₩0'이 없어서 무한히 다음 주소를 참조하는 상태가 되어 시스템이 다운되었다. 따라서 두 문자를 입력받은 문자열의 길이만큼만 비교하는 strncmp 함수를 이용하여 문자의 끝을 나타내는 값이 없더라도 메모리 참조 범위를 넘어가지 않도록 했으며, 두 char 모두 주소를 참조하기 때문에 이 둘의 주소가 NULL인지를 먼저 체크하여 예외처리를 했다.

```
if(hw1_file_system_type[i])
if(user_buffer)
if(strncmp(hw1_file_system_type[i], user_buffer, count-1) == 0)
if(hw1_buffer[i]!=0) {
    // ----- added by Inho (16. 10. 26) ----- //
    printk(KERN_INFO "%d:[%lld] %lld\n", i, hw1_time[i], hw1_buffer[i]);

    // ----- added by Inho (16. 10. 26) ----- //
    snprintf(tmp, 19, "%lld", hw1_time[i]);
    // ----- added by Hakyoung (16. 10. 27) ----- //
    vfs_write(filp, tmp, strlen(tmp), &filp->f_pos);
    vfs_write(filp, ", ", 2, &filp->f_pos);

    // ----- added by Inho (16. 10. 26) ----- //
    snprintf(tmp, 19, "%lld", hw1_buffer[i]);

    // ----- added by Hakyoung (16. 10. 27) ----- //
    vfs_write(filp, tmp, strlen(tmp), &filp->f_pos);
    vfs_write(filp, ", ", 2, &filp->f_pos);

    // ----- added by Inho (16. 10. 26) ----- //
    printk("%s\n", hw1_file_system_type[i]);

    // ----- added by Hakyoung (16. 10. 27) ----- //
    snprintf(tmp, 19, "%s", hw1_file_system_type[i]);
    vfs_write(filp, tmp, strlen(tmp), &filp->f_pos);
    vfs_write(filp, "\n", 1, &filp->f_pos);

    hw1_time[i]=0;
    hw1_buffer[i]=0;
    hw1_file_system_type[i]=NULL;
}
```

3. 결과

실행 방법 및 실행 결과

① 커널 코드인 blk-core.c, segbuf.c를 ReadMe.txt에 언급된 경로에 저장한 뒤 컴파일하고 커널을 설치한다. (수정된 부분은 // ---- added ---- 형식으로 주석이 달려 있다.)

② hw1.ko 모듈을 insmod 명령어를 통해 커널에 삽입한다.

③ 쓰기 실험을 할 파일시스템별로 각각 가상디스크를 만들어 포맷한 후 I/O device로 등록하여 서로 다른 디렉토리에 마운트한다.

④ 각 파일시스템 디렉토리로 접근하여 iozone -a 0 명령어를 통해 write를 수행한다.

⑤ /proc/myproc/ 경로로 접근하여 echo 파일시스템명 > hw1 명령어를 수행한다. 파일시스

템명에는 출력하고 싶은 파일시스템의 이름을 입력한다. ext4, nilfs2 둘 중에 하나를 입력하면 된다. 제대로 된 파일시스템명을 입력하지 않는 경우에는 어떤 것도 출력되지 않는다.

⑥ 출력된 값은 dmesg 명령어, 또는 /tmp/ 에 생성된 result.csv 파일로 확인 가능하다. 시간과 sector number, 파일시스템명 순으로 출력된다.

dmesg 출력 결과

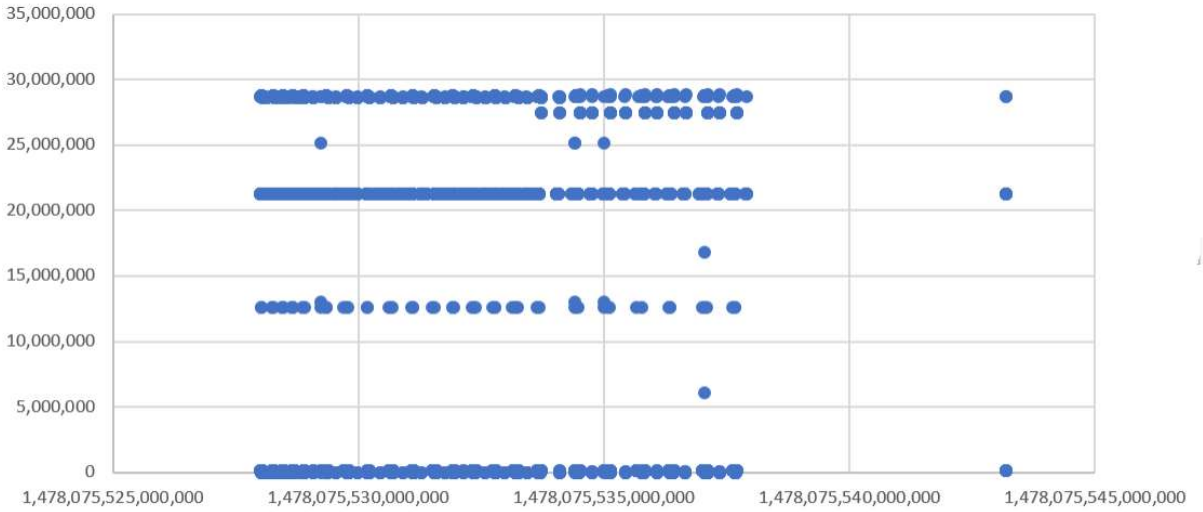
```
[  974.166293] ext4                                  [  843.553600] nilfs2
[  974.166297] 764:[1478075537114816] 27475968       [  843.553605] 291:[1478075406781932] 118256
[  974.166303] ext4                                  [  843.553611] nilfs2
[  974.166306] 765:[1478075537114969] 27478016       [  843.553614] 292:[1478075406783055] 120304
[  974.166312] ext4                                  [  843.553620] nilfs2
[  974.166316] 766:[1478075537115082] 27480064       [  843.553624] 293:[1478075406783583] 122352
[  974.166322] ext4                                  [  843.553630] nilfs2
[  974.166326] 767:[1478075537115189] 27482112       [  843.553633] 300:[1478075406845992] 122544
[  974.166331] ext4                                  [  843.553639] nilfs2
[  974.166335] 768:[1478075537115294] 27484160       [  843.553643] 301:[1478075406846657] 124592
[  974.166341] ext4                                  [  843.553649] nilfs2
[  974.166344] 769:[1478075537115416] 27486208       [  843.553652] 302:[1478075406847165] 126640
[  974.166350] ext4                                  [  843.553658] nilfs2
[  974.166354] 770:[1478075537115524] 27488256       [  843.553662] 310:[1478075406947956] 126840
[  974.166360] ext4                                  [  843.553668] nilfs2
[  974.166363] 771:[1478075537115630] 27490304       [  843.553672] 316:[1478075406984851] 126992
[  974.166369] ext4                                  [  843.553677] nilfs2
[  974.166373] 772:[1478075537115636] 28819456       [  843.553681] 317:[1478075406986186] 129040
[  974.166379] ext4                                  [  843.553687] nilfs2
[  974.166382] 773:[1478075537115638] 28691168       [  843.553691] 318:[1478075406986927] 131072
[  974.166388] ext4                                  [  843.553696] nilfs2
[  974.166392] 774:[1478075537312827] 21279112       [  843.553700] 325:[1478075407049232] 131288
[  974.166398] ext4                                  [  843.553706] nilfs2
[  974.166401] 775:[1478075537312831] 21279120       [  843.553709] 326:[1478075407049269] 133336
```

result.csv 출력 결과

```
1478075533698921, 20482, ext4
1478075533699102, 22530, ext4
1478075533699412, 24578, ext4
1478075533699574, 26626, ext4
1478075533699873, 28674, ext4
1478075533700032, 30722, ext4
1478075533700413, 32770, ext4
1478075533700599, 34818, ext4
1478075533724683, 182776, ext4
1478075533724688, 182778, ext4
1478075533724688, 182780, ext4
1478075533724689, 182782, ext4
1478075533724689, 182784, ext4
1478075533724721, 182786, ext4
1478075533724903, 28643328, ext4
1478075533725026, 28645376, ext4
1478075533725134, 28647424, ext4
1478075533725256, 28649472, ext4
1478075533725456, 28651520, ext4
1478075533725562, 28653568, ext4
1478075533725809, 27475968, ext4
1478075533725915, 27478016, ext4
1478075533725921, 27480064, ext4
```

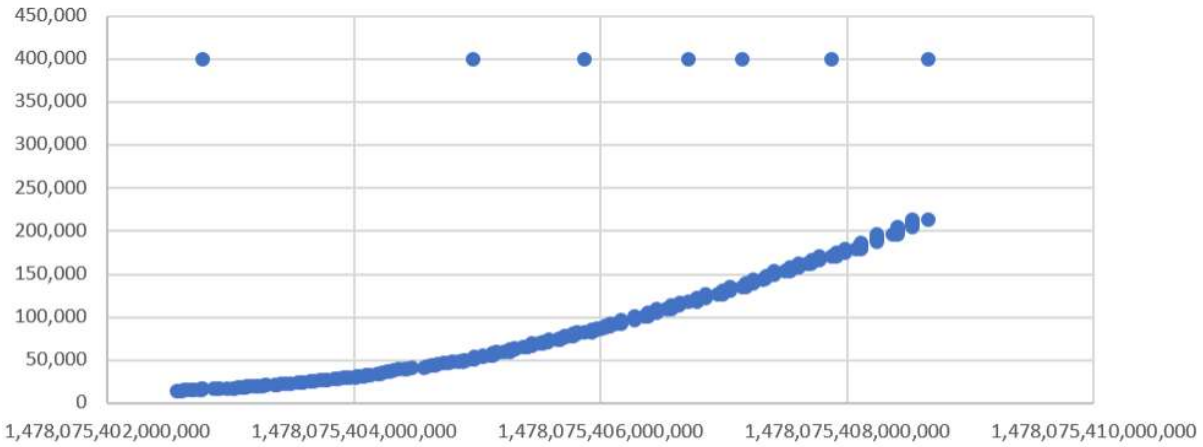
결과 그래프 및 분석

1. ext4



write 작업이 이루어지는 디스크 섹터가 순차적으로 이어지지 않고 멀리 떨어져 있는 섹터에의 접근이 발생하는 것을 관찰할 수 있다. 그 이유는 ext4에서의 파일 구조가 super block, inode, data block을 디스크 상에 서로 다른 위치에 저장한 뒤에 맵핑 테이블을 통해서 참조하는 식으로 이루어져 있기 때문이다. 특히 0에 가까운 섹터 번호에 지속적으로 write가 발생하는 것은 super block에 접근을 하기 때문이라고 추정된다.

2. nilfs2



ext4와 확연한 차이를 보인다. 시간에 따라 연속된 섹터에 순차적으로 접근하는 것을 관찰할 수 있다. 이는 nilfs2가 Log Structured File System을 기반으로 하기 때문이다. write 과정에서 메타데이터와 실제 data block을 개별적으로 저장해두고 참조하는 방식이 아니라, 연속된 공간에 순차적으로 접근을 하여 write을 수행한다.