COSE322-00
시스템프로그래밍
유혁 교수님

1차 과제

Log Structured File System Profiling - ext4와 비교

제출일: 2019.10.31 (프리데이 사용일수: 0)
개발환경:
가상머신: Oracle VM VirtualBox
운영체제: Ubuntu 16.04 LTS
커널: linux-4.4.0

조장: 2015131102 강수진
조원: 2015131116 배은초

목차

- nilfs2 write log file

1. 들어가기 전에

배경 지식

1. Log-structured file system이란?

Log-structre file system이란 meta data와 data가 비동기적으로 디스크에 작성된다는 점에서 착안되었다. 기존의 파일 시스템의 경우, write 발생 시, 메타 데이터는 발생과 동시에 업데이트 되지만 데이터의 경우 캐시에 저장되어 있다가 한 번에 디스크에 작성된다. 또 메타 데이터와 데이터가 디스크의 다른 섹터에 저장되므로 디스크에 쓰기를 수행할 때 작은 쓰기들이 여러 번 발생하게 되는데, 작은 쓰기를 할 경우 매 쓰기마다 seek time이 소요되므로 효율적이지 못하다. 또 쓰기 동작 수행중 crash가 일어나면, 모든 데이터 블록의 inode를 스캔하여 어디까지 update되었는지 확인해야 하는 등 recovery가 복잡하고 어렵다.

이와 다르게 로그 기반 파일시스템은 데이터블록, inode, inode map등을 일련적으로 디스크에 저장한다. 모든 정보를 한 번에 모아서 처리하기 대문에 작은 쓰기에서 발생하는 seek time을 없앨 수 잇다. Crash recovery의 경우에도 파일의 제일 끝 부분의 정보만 살펴서 crash가 일어난 부분을 찾고 recover할 수 있다는 점에서 더 효율적이다. 하지만 inode와 data가 무질서하게 혼재되어 작성되기 때문에 read 시에 inode를 찾기가 어렵다. 그래서 파일에 번호를 부여하고, 간단한 계산으로 inode의 위치를 파악할 수 있도록 inode map을 도입했다. Inode map은 메모리에 캐싱되어 있으며 기존 파일 시스템과 구별되는 로그 기반 파일시스템의 특징적인 부분이다.

물론 로그 기반 파일 시스템도 단점이 존재한다. 게속하여 파일의 뒤에 append하기 때문에 시간이 지남에 따라 디스크 공간이 부족해질 수 있다. 따라서 빈 공간을 확보하는 garbage collection이 필요하다.

로그 파일 시스템은 garbage collection이 필요하고, inode map이 필요하다는 overhead가 있지만 이런 단점들은 flash에선 크게 부각되지 않는다. 플래시의 경우, wear-leveling의 문제때문에 쓰기 수행 시 기존 위치가 아닌 새로운 공간에 (가장 사용되지 않은 공간에) 쓰기를 새로 수행한다. 따라서 플래시에서 garbage collection은 필수적이기 때문이다.

2. Nilfs2란

Nilfs2는 로그 기반 파일 시스템의 한 종류이다. 저장 매체가 circular buffer처럼 사용되며 새로운 블럭은 항상 끝에 쓰인다. Nilfs는 스냅샷이라는 개념을 사용하는데 스냅샷은 일종의 쓰기 동작의 복사본같은 역할을 한다. 쓰기 동작 수행시 생성되는 지속적인 스냅샷들은 유저로부터 시작된 파일시스템 오류에서 복구하는 데 도움이 된다. Nilfs는 몇 초 혹은 동기화된 쓰기를 한 번 시행시 체크 포인트를 만들어 내는데, 유저는 이 체크포인트를 스냅샷으로 플래깅한다. 순간적인 상태를 저장한 체크포인트들은 lscp 커맨드를 통해 볼 수 있다.

Nilfs2는 로그 기반 파일 시스템의 한 종류이기 때문에, 로그 기반 파일 시스템의 특성을 가진다. 모든 데이터를 연속적인 로그 포맷으로 저장하기 때문에 이미 쓰인 곳에 다시 쓰이지 않으며, 이러한 방식으로 인해 seek time이 작으며, 데이터 소실이 적게 일어난다. 다시 말해 쓰기 속도가 기존의 파일 시스템보다 빠르다. 하지만 스냅샷이 로그 형태여서 예전에 쓰인 데이터가 남아있다는 점, 이로 인해 가비지 컬렉션이 필요하다는 점, 메타데이터가 늘어난다는 점 때문에 성능 저하의 문제가 일어날 수도 있다.

3. Fast File System 이란?

File system은 boot block, super block, inode, data block 등 4개의 중요한 부분으로 구성되어 있다. 파티션의 처음 부분에 존재하는 부트 블럭들은 파일 시스템과 분리되어 시작되어야 한다. 수퍼블럭은 은 파일 시스템을 구분하는 번호와 그 외 파일 시스템에 대한 중요한 정보를 가진다. 아이노드는 각 파일에 대한 정보를 담고 있고, 데이터 블럭은 실제 데이터를 갖고 있다. 초기 File system은 이 네가지로만 구성되었지만 이후 디스크가 커지면서 헤더를 아이노드와 데이터 블럭 간 앞뒤로 움직이는 것이 힘들어졌다. 또 기존 파일 시스템은 블록 사이즈가 작아서 디스크 bandwidth를 충분히 사용하지 못했다. 따라서 이를 보완하기 위해 Fast file system이 도입되었다.

Fast File system이 기존의 파일 시스템과 다른 점은 실린더 그룹이 존재한다는 것이다. 실린더 그룹엔 superblock의 백업본, 실린더 그룹 헤더(실린더 그룹에 대한 정보 가짐), inode, data block이 들어있다. 이렇게 함으로써 한 디렉토리에 대한 데이터, 메타데이터가 물리적으로 가까운 곳에 위치하게 되어 한 디렉토리의 데이터들이 전체 디스크에 흩어져 있을 때보다 효율적이게 되었다. 이외에도 fast file system은 블록 사이즈를 늘렸고, fragmentation을 통해 블럭을 나누어 낭비되는 공간을 줄였다.

4. Ext4란?

ext4는 ext가 발전해 나온 파일 시스템이다. 새로운 파일 시스템을 더하기 쉽도록 리눅스는 VFS layer를 도입하였는데, 이에 맞추어 VFS API를 사용한 첫 파일 시스템인 ext가 탄생하였다. ext란 'extended file system'의 약자로 1992년 리눅스에 추가되었다. ext는 기존 파일 시스팀(MINIX file system)의 파일 최대 크기가 64MB이고 파일명을 14자밖에 쓰지 못한다는 한계를 극복했다.

ext에서 더 발전한 ext2는 FFS중에서도 Berkeley fast file system의 원칙을 도입했다. 따라서 위에서 설명한 Fast File System의 특성을 지닌다. ext2 파일시스템에서 디스크는 블록으로 나누고, 블록들은 블록 그룹으로 묶는다. 블록 그룹은 unix(fast) file system

의 실린더 그룹과 비슷하다. 각 블록 그룹은 그룹 서술 테이블과 수퍼블록 백업을 포함하며, 블록 비트맵, 아이노드 비트맵, 아이노드 테이블, 실제 데이터 블록을 가지고 있다. 하지만 ext2는 시스템 크래쉬나 갑작스럽게 전원이 끊기는 경우 파일시스템이 심각하게 손상되었다. 또 데이터를 디스크에 분산 저장하기 때문에 fragmentation 때문에 성능이 저하될 수 있었다.

ext2의 이런 문제점을 해결하기 위해 ext3는 journaling을 적용했다. 저널이란 디스크에 할당된 특정 공간으로 한 쓰기 트랜잭션을 완료하면 저널안의 데이터를 commit하게 된다. 만약 commit 전에 crash가 발생하거나, 전원이 꺼지면 시스템은 저널영역만 없애고 기존 파일시스템은 그대로 남긴다. 이렇게 되면 파일은 손상되지만 파일 시스템은 손상되지 않게 되어 다른 데이터는 안전하게 유지할 수 있다.

ext4는 ext3에서 몇 가지가 더 발전했는데, 우선 ext3보다 큰 용량의 파일 시스템을 지원하게 되었다. 또, extent를 사용하는데, extent란 한 번에 처리할 수 있는 물리적 블록의 범위이다. extent를 사용하면 inode의 수, framentation 등을 줄일 수 있다. 지연할당 기능도 추가되었는데, ext4의 경우 다른 파일 시스템과 달리 데이터가 디스크에 쓰일때까지 블록 할당이 지연된다. 지연 할당을 통해 성능이 더 좋아지고, 실제 파일 크기에 맞춰 공간을 할당할 수 있어 데이터가 조각나는 것을 막을 수 있다. 이외에도 서브 디렉토리 제한이 없어지고 저널링에서 발생하는 파일 손상을 알 수 있도록 하는 체크섬을 도입하였다.

역할 분담

강수진

- Queue 구조체 구현
- LKM 작성
- iozone 통한 disk 성능 측정
- 데이터 가공
- 데이터 plotting

배은초

- 환경 세팅
- blk-core.c에서 구조체 분석 및 수정 통해 파일 시스템의 이름, 시간, sector 번호 접근
- segbuf.c 코드 수정 통해 nilfs2 환경에서도 위의 동작을 가능하게 만듦
- null exception 등에 따른 예외 처리
- 컴파일 오류 처리

2. 개발

Kernel Code

bio-core.c 의 `blk_qc_t submit_bio(int rw, struct bio *bio)` 함수 안에서 write할때 각각의 파일 시스템 이름과, 시간, sector 번호를 출력하면 된다. `if (rw & WRITE)` condition 안에서 각 값을 구하는 로직을 작성하면 된다.

해당 값들이 push되어 있는 queue를 `EXPORT_SYMBOL` 을 통해 다른 곳에서도 사용할 수 있게 하고, 이후 pop을 통해 각 값을 프린트한다.

파일 시스템 이름 구하기

파일 시스템 이름을 참조하기 위해서는 super block 구조체 안의 여러 구조체들을 참조해 들어가야 한다. 각 파일 시스템은 generic한 레이어로 `submit_bio()`를 통해 공통적인 구조체 형식인 bio를 전해, 이를 통해 read/write를 수행할 수 있게 한다. 유저 영역에서 파일 시스템 명을 참조하기 위해서는 bio가 파일시스템 명을 참조하게 해야 하기 때문에 bio의 구조를 연구했다.

그 결과 알게 된 사실은 다음과 같다:

`name`을 참조하려면 `super_block` 구조체의 `s_type->name`를 타고 들어가야한다.

`block_device`는 `bd_super` 라는 이름으로 `super_block` 구조체를 참조한다.

`bio`는 `bi_bdev`라는 이름으로 `block_device` 구조체를 참조한다.

따라서 `submit_bio(int rw, struct bio *bio)` 함수에서 해당 `name`을 참조하기 위해서는 아래와 같이 참조할 수 있다.

```
bio->bi_bdev->bd_super->s_type->name;
```

이후 ext4와 nilfs2 에서 `submit_bio()`가 어떻게 호출되는지 알아보았다.

EXT4

ext4에서는 `submit_bio`를 호출하기 전 어떤 동작이 일어나는지 `readpage.c` 를 통해 살펴 보았다.

/linux/fs/ext4/readpage.c

해당 파일 안에는 아래와 같은 함수가 있다. 해당 함수 안에서는 submit_bio(READ, bio); 와 같은 형태로 submit_bio 함수를 호출한다.

```
int ext4_mpage_readpages(struct address_space *mapping,
       struct list_head *pages, struct page *page,
       unsigned nr_pages, bool is_readahead)
```

이때 인자로 넘겨줄 bio가 NULL이면 bio_alloc을 통해 객체를 만들어주는데 이때 bio_set_dev(bio, bdev); 함수를 통해 해당 bio와 bdev set 해준다.

```
if (bio == NULL) {
      struct bio_post_read_ctx *ctx;

      bio = bio_alloc(GFP_KERNEL,
        min_t(int, nr_pages, BIO_MAX_PAGES));
      if (!bio)
        goto set_error_page;
      ctx = get_bio_post_read_ctx(inode, bio, page->index);
      if (IS_ERR(ctx)) {
        bio_put(bio);
        bio = NULL;
        goto set_error_page;
      }
      //여기서 bio와 bdev의 정보를 연결.
      bio_set_dev(bio, bdev);
      bio->bi_iter.bi_sector = blocks[0] << (blkbits - 9);
      bio->bi_end_io = mpage_end_io;
      bio->bi_private = ctx;
      bio_set_op_attrs(bio, REQ_OP_READ,
            is_readahead ? REQ_RAHEAD : 0);
    }
```

이때 bdev는 ext4_mpage_readpages 함수 안에서 선언된 변수로서, 함수의 parameter로 들어온 mapping(address_space 타입) 정보를 통해 값이 할당된다.

```
// ext4_mpage_readpages 함수 안에는 아래와 같은 변수들이 있음
struct bio *bio = NULL;
//mapping은 해당 함수가 불릴때 넘어온 parameter
struct inode *inode = mapping->host;
struct block_device *bdev = inode->i_sb->s_bdexreev;
```

이렇게 bdev는 super_block 타입인 i_sb의 정보까지 갖게 되고, 이를 bio와 매칭시켜주면 bio→bi_bdev→bd_super→s_type->name; 과 같이 이름을 접근할 수 있다.

Nilfs2

/linux/fs/nilfs2/segbuf.c

nilfs2에서 submit_bio()를 호출하는 과정은 좀 달랐다. Nilfs2에서 submit_bio를 호출하는 함수는 nilfs_segbuf_submit_bio() 로 이 함수에는 bio안의 bi_bdev(block device 구조체)내에 super_block 구조체를 별도로 할당하고 있지 않고 있었다.

```
//nilfs에서 submit_bio를 호출하는 함수
static int nilfs_segbuf_submit_bio(struct nilfs_segment_buffer *segbuf,
           struct nilfs_write_info *wi, int mode)
{
  struct bio *bio = wi->bio;
  int err;
```

```
//생략

  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++;
```

따라서 bi_bdev내의 super_block 구조체인 bd_super에 segbuf (nilfs_segment_buffer 구조체)에 들어있는 super_bd(super_block구조체)를 넣어주었다. 이렇게 함으로써 이후 nilfs2에서도 파일시스템명을 참조할 수 있게 되었다.

참고) segbuf 구조체 정의:

```
// segbuf의 구조체 정의, 안에 superblock pointer 있음
struct nilfs_segment_buffer {
  struct super_block     *sb_super;
```

소결론

결국 둘의 차이는 submit_bio 호출하기 전에 각 파일시스템에서 bio를 어떻게 세팅하냐의 문제였다. bio가 superblock을 참조할 수 있어야 하는데 ext4는 자체적으로 파일시스템 내에서 super block을 참조할 수 있도록 하지만, nilfs는 그런 기능이 없기 때문에 따로 bio→bi_bdev→bd_super에 넣어주어야 했다.

시간 구하기

아래 코드를 통해 시간을 얻어올 수 있다.

```
struct timeval mytime;
do_gettimeofday(&mytime);
(unsigned long long)(mytime.tv_sec) * 1000000 + (unsigned long long)(mytime.tv_usec);
```

커널 영역에서는 유저 영역에서처럼 wall-clock 시간으로 하는 건 좋지 않으므로 커널 영역의 절대 timestamp로 시간을 받아옴. 절대 시간을 찍기 위해 <linux/time.h> 를 include하여 do_gettimeofday 함수를 사용한다. 이 함수로 인자로 넘긴 timeval 포인터를 채워준다. timval 구조체 내의 tv_sec은 그 시간의 초, tv_usec은 그 순간의 마이크로초이므로 tv_sec*100000 + tv_usec을 해서 마이크로초 단위로 맞춰준다.

참고) timeval 구조체

```
struct timeval
{
  long tv_sec; //초
  long tv_usec;  //마이크로초
}
```

sector 번호 구하기

아래 코드를 통해 sector을 얻어올 수 있다.

```
(unsigned long long) bio->bi_iter.bi_sector
```

적용하기

먼저 if문을 통해 NULL이 될 수 있는 있는 부분에 대한 exception 처리를 한다. 그 후 위에서 설명한대로 시간, sector 번호, 파일 시스템 이름을 구하고 각 값을 통해 Process라는 커스텀한 구조체를 따르는 인스턴스를 만들어 priority queue에 넣었다. 이때 priority 기준은 도착 시간이다.

```
if(bio->bi_iter.bi_sector!=NULL) if(bio->bi_bdev != NULL) if(bio->bi_bdev->bd_super != NULL) if(bio->bi_bdev->bd_super->s_type
                do_gettimeofday(&mytime);
                //생성 후 넣어줌. 순서대로 time, sector, system type name
                Process temp = createProcess((unsigned long long)(mytime.tv_sec) * 1000000 + (unsigned long long)(mytime.tv_use
                                             (unsigned long long) bio->bi_iter.bi_sector,
                                             bio->bi_bdev->bd_super->s_type->name);
                pq_push(&jobQueue, temp);
                }
```

참고

Priority Queue 구현 (arrival time 순)

`swap` 이라는 함수와 `current`란 변수를 따로 정의해서 사용했는데 컴파일 중 에러가 생겼다. 전자는 이미 blk-core.c 파일에 `#include<swap.h>`에 정의되어 있어서 같은 형식의 함수가 그런 듯 했다. 후자도 비슷하게 `current` 이름의 변수가 예약어로 사용되고 있는듯 했다. 따라서 각각의 이름을 `my_swap`과 `my_current`로 바꾸어 사용하였다.

```
struct timeval mytime;
#define MAX_SIZE 2000
typedef struct {
    unsigned long long sector;
    long long int arrival_time;
    const char* system_type;
} Process;


typedef struct priority_queue {
    Process heap[MAX_SIZE];
    int size;
} priority_queue;

priority_queue jobQueue = {.size = 0};
EXPORT_SYMBOL(jobQueue);

//////////////Process//////////////
Process createProcess(long long int arrival_time, unsigned long long sector, const char*  system_type) {
    Process process = {.arrival_time = arrival_time, .sector = sector, .system_type = system_type};
    return process;
}

//////////////Queue/////////////////

void my_swap(Process *a, Process *b)
{
    Process tmp = *a;
    *a = *b;
    *b = tmp;
}

Process pq_pop(priority_queue* q) {
    int my_current = 0;
    int leftChild = my_current * 2 + 1;
    int rightChild = my_current * 2 + 2;
    int maxNode = my_current;
    //비어있으면 return
    Process process = {.arrival_time = -1};
    if (q->size <= 0) return process;

    //우선 순위 큐에서 pop 할 것은 가장 우선 순위가 높은 노드, 즉 루트
    Process ret = q->heap[0];
    q->size--;

    //재정렬
    q->heap[0] = q->heap[q->size]; //루트에 가장 낮은거 올림

    while (leftChild < q->size) {
        //left child 가 있는데 max node의 arrival time 이 더 큰 경우
        if ((q->heap[maxNode]).arrival_time > (q->heap[leftChild]).arrival_time) {
            maxNode = leftChild;
        }
        //right child 까지 있는데 max node(방금전까지 leftChild의 값)의 arrival time 이 더 큰 경우
        if (rightChild < q->size && q->heap[maxNode].arrival_time > q->heap[rightChild].arrival_time) {
            maxNode = rightChild;
        }

        if (maxNode == my_current) {
            break;
        }
        else {
            my_swap(&(q->heap[my_current]), &(q->heap[maxNode]));
```

```
            my_current = maxNode;
            leftChild = my_current * 2 + 1;
            rightChild = my_current * 2 + 2;
        }
    }
    return ret;
}
//밖에서 써줘야하니까 export
EXPORT_SYMBOL(pq_pop);

int pq_push(priority_queue* q, Process value) {
    int size = q -> size;

  //다 차면 circular queue처럼 동작하도록
    if (size + 1 > MAX_SIZE) {
       pq_pop(q);
       size--;
    }

    int my_current = size; //현재 위치한 인덱스
    int parent = (size - 1) / 2; //완전 이진트리에서 parent 인덱스

    q -> heap[size] = value; //마지막 빈 자리에 value 할당

    //재정렬
    while (current > 0 && (q ->heap[my_current].arrival_time) < (q ->heap[parent].arrival_time)) {
        my_swap(&(q->heap[my_current]), &(q ->heap[parent]));
        my_current = parent;
        parent = (parent - 1) / 2;
    }


    q ->size++;
    return 1;
}
```

고친 부분 모아보기

/linux/block/blk-core.c

36 line

```
#include <linux/time.h>
```

53 line - Queue 구현부

```
struct timeval mytime;
#define MAX_SIZE 2000
typedef struct {
    unsigned long long sector;
    long long int arrival_time;
    const char* system_type;
} Process;


typedef struct priority_queue {
    Process heap[MAX_SIZE];
    int size;
} priority_queue;

priority_queue jobQueue = {.size = 0};
EXPORT_SYMBOL(jobQueue);

//////////////Process//////////////
Process createProcess(long long int arrival_time, unsigned long long sector, const char*  system_type) {
    Process process = {.arrival_time = arrival_time, .sector = sector, .system_type = system_type};
    return process;
}

//////////////Queue/////////////////

void my_swap(Process *a, Process *b)
{
    Process tmp = *a;
    *a = *b;
    *b = tmp;
}

Process pq_pop(priority_queue* q) {
    int my_current = 0;
    int leftChild = my_current * 2 + 1;
```

```
    int rightChild = my_current * 2 + 2;
    int maxNode = my_current;
    //비어있으면 return
    Process process = {.arrival_time = -1};
    if (q->size <= 0) return process;

    //우선 순위 큐에서 pop 할 것은 가장 우선 순위가 높은 노드, 즉 루트
    Process ret = q->heap[0];
    q->size--;

    //재정렬
    q->heap[0] = q->heap[q->size]; //루트에 가장 낮은거 올림

    while (leftChild < q->size) {
        //left child 가 있는데 max node의 arrival time 이 더 큰 경우
        if ((q->heap[maxNode]).arrival_time > (q->heap[leftChild]).arrival_time) {
            maxNode = leftChild;
        }
        //right child 까지 있는데 max node(방금전까지 leftChild의 값)의 arrival time 이 더 큰 경우
        if (rightChild < q->size && q->heap[maxNode].arrival_time > q->heap[rightChild].arrival_time) {
            maxNode = rightChild;
        }

        if (maxNode == my_current) {
            break;
        }
        else {
            my_swap(&(q->heap[my_current]), &(q->heap[maxNode]));
            my_current = maxNode;
            leftChild = my_current * 2 + 1;
            rightChild = my_current * 2 + 2;
        }
    }
    return ret;
}
//밖에서 써줘야하니까 export
EXPORT_SYMBOL(pq_pop);

int pq_push(priority_queue* q, Process value) {
    int size = q -> size;

  //다 차면 circular queue처럼 동작하도록
    if (size + 1 > MAX_SIZE) {
       pq_pop(q);
       size--;
    }

    int my_current = size; //현재 위치한 인덱스
    int parent = (size - 1) / 2; //완전 이진트리에서 parent 인덱스

    q -> heap[size] = value; //마지막 빈 자리에 value 할당

    //재정렬
    while (current > 0 && (q ->heap[my_current].arrival_time) < (q ->heap[parent].arrival_time)) {
        my_swap(&(q->heap[my_current]), &(q ->heap[parent]));
        my_current = parent;
        parent = (parent - 1) / 2;
    }


    q ->size++;
    return 1;
}
```

2214 line - submit_bio 함수 안 time, sector번호, system이름 구하기

```
if(bio->bi_iter.bi_sector!=NULL) if(bio->bi_bdev != NULL) if(bio->bi_bdev->bd_super != NULL) if(bio->bi_bdev->bd_super->s_type

                //생성 후 넣어줌
                Process temp = createProcess((unsigned long long)(mytime.tv_sec) * 1000000 + (unsigned long long)(mytime.tv_use
                                             (unsigned long long) bio->bi_iter.bi_sector,
                                             bio->bi_bdev->bd_super->s_type->name);
                pq_push(&jobQueue, temp);
                }
```

/linux/fs/nilfs2/segbuf.c

372 line

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

코드 파일

blk-core.c (원래 경로 - /linux/block/blk-core.c)

/KernelCode/blk-core.c

'added by' 로 수정 부분 검색 가능

segbuf.c (원래 경로 - /linux/fs/nilfs2/segbuf.c)

/KernelCode/segbuf.c

'added by' 로 수정 부분 검색 가능

LKM

PROC

새로운 모듈에서 proc_create()를 통해 proc_dir_entry 구조체를 만들 수 있다.

이때 proc_dir_entry에는 *proc_fops(struct file_operations)가 있고 write나 open에 대한 함수 포인터를 확인 할 수 있다.

```
struct proc_dir_entry {
  const struct file_operations *proc_fops;
}
```

```
struct file_operations {
  ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
  int (*open) (struct inode *, struct file *);
}
```

proc_create를 통해 proc_dir_entry를 만들때 네번째 인자로 커스텀한 file_operations을 넣어줌으로써 custom한 동작을 지정할 수 있다.

```
static struct proc_dir_entry *proc_file;
proc_file = proc_create("proc_file_name", 0600, proc_dir, &custom_proc_fops);

static const struct file_operations custom_proc_fops = {
    .owner = THIS_MODULE,
    .write = custom_write,
};
```

Custom Write 함수 구현

```
#define MAX_SIZE 2000

typedef struct {
    unsigned long long sector;
    long long int arrival_time;
    const char* system_type;
} Process;

typedef struct priority_queue {
    Process heap[MAX_SIZE];
    int size;
} priority_queue;

extern priority_queue jobQueue;
extern Process pq_pop(priority_queue*);

static ssize_t custom_write(struct file *file, const char __user *user_buffer, size_t count, loff_t *ppos) {

        printk(KERN_ALERT "write!\n");

    while (jobQueue.size != 0) {
        Process popped = pq_pop(&jobQueue);
        int arrivalTime = popped.arrival_time;
        if (arrivalTime == -1) {
             //다 뺐는데 읽으려고하면 return
            return 0;
        }
```

```
        printk(KERN_INFO ", %s, %lld, %lld\n", popped.system_type, popped.arrival_time, popped.sector);
    }

    printk(KERN_ALERT "write complete.\n");
    return count;
}
```

printk의 형식을 위와 같이 한 이유(fileSystem, time, sector)는 썼을때 csv 형식으로 뽑아내기 쉽게 하기 위해서이다.

참고

코드 파일

linux kernel module

/LKM/mybasic.c

/LKM/mybasic.ko

Makefile

/LKM/Makefile

3. 결과

실행 방법

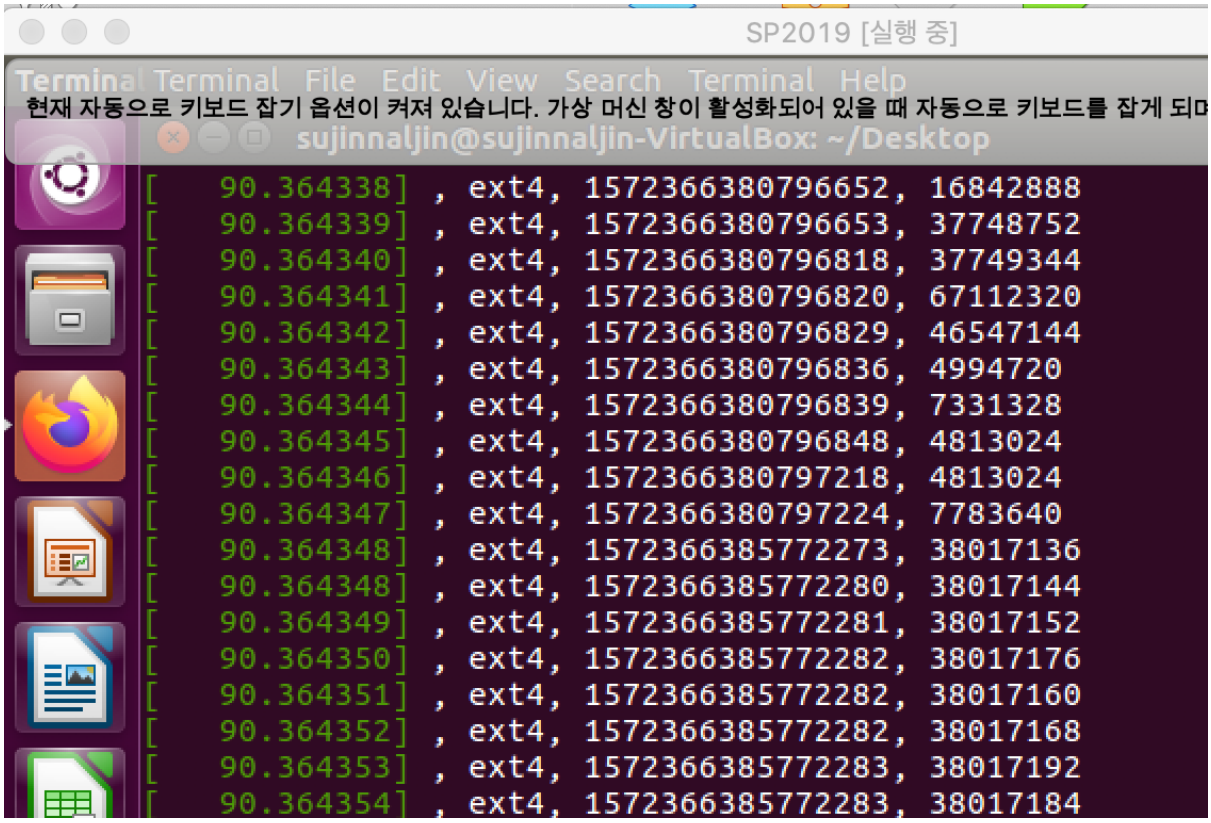
1. 원래 커널 소스에 있는 blk-core.c와 segbuf.c 우리가 작성한 blk-core.c와 segbuf.c파일로 대체
2. 커널 컴파일 후 해당 커널로 부팅

```
uname -r //현재 커널 버전 확인
vi /boot/grub/grub.cfg //여기서 menu_entry, sub_entry 중에 원하는 커널 버전 확인. menu_entry는 0, sub_entry는 1>0, 1>1, .. 1>n
vi /etc/default/grub //GRUB-DEFUALT 값 확인 후 원하는 값으로 변경 ex GRUB-DEFUALT = "1>3"
sudo update-grub
reboot
uname -r //부팅된 커널 버전 확인
```

3. Linux Kernel Module 작성 → mybasic.c 에 해당
4. Makefile 작성. 이때 KDIR는 커널 소스 경로.
5. make 명령어 실행
6. 커널 모듈 올리기 → sudo insmod basic.ko
7. 수행하고자 하는 fs가 mount 된 dir 로 이동
8. 그 디렉토리에 write 수행할 파일을 생성. sudo vim temp
9. 같은 디렉토리에서 iozone -a -f ./temp -명령어를 통해 temp 파일에 write 수행
10. cd /proc/myproc_dir/ 통해 경로 이동
11. sudo bash -c 'echo 문자 > myproc_file' 명령어 수행
12. myproc_dir 에서 dmesg를 통해 출력 결과 확인

결과 시각화 및 해석

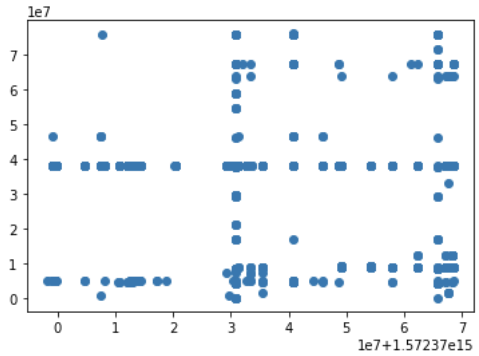
각 파일시스템 디렉토리로 접근하여 iozone을 통해 write를 수행하고 dmesg를 통해 로그를 확인하니 다음과 같았다. (사진은 ext4 상에서 수행한 것)



각각의 로그를 .txt 파일로 빼내서 jupyter notebook을 통해 scatter했다. 이때 nilfs2에 해당하는 로그만 빼내기 위해서 jupyter notebook을 통해 preprocessing 하는 과정 또한 거쳤다. (log_preprocessing.ipynb)

EXT4

```
df = pd.read_csv("./iozone_log_ext4.txt", names = ['title', 'systemType', 'time', 'sector'])
plt.scatter(df['time'], df['sector']);
```

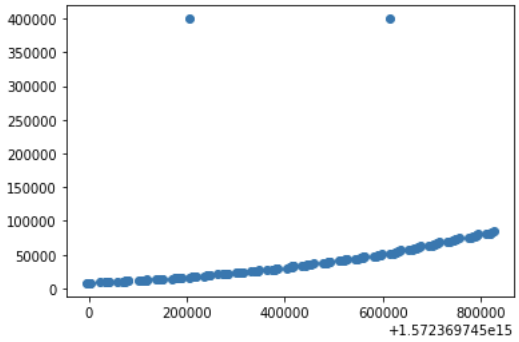


/Data & Preprocessing & Plotting/ext4_plotting.png

EXT4는 FFS을 기반으로 동작한다. super block, data block, inode 등을 각각 다른 위치에 저장하고 매핑 테이블을 통해 참조한다. 따라서 write 수행 시 멀리 떨어져 섹터에도 접근이 발생한다. plotting된 log 데이터에서도 실제로 이와 같은 특성을 관찰할 수 있다.

NILFS2

```
df = pd.read_csv("./iozone_log_edited_nilfs2.txt", names = ['title', 'systemType', 'time', 'sector'])
plt.scatter(df['time'], df['sector']);
```



/Data & Preprocessing & Plotting/nilfs2_plotting.png

NILFS2는 Log Structured File System을 기반으로 동작한다. 'Write all the modifications sequentially'를 idea로 삼아, LFS는 write 수행 시 연속된 섹터에 순차적으로 접근한다. plotting된 log 데이터에서도 실제로 이와 같은 특성을 관찰할 수 있다.

참고

코드 파일

preprocessing data using jupyter notebook (only for nilfs2)

/Data & Preprocessing & Plotting/log_preprocessing.ipynb

plotting data using jupyter notebook

/Data & Preprocessing & Plotting/plotting_fs.ipynb

로그 파일

ext4&nillfs2 write log file

/Data & Preprocessing & Plotting/iozone_log_ext4&nilfs2.txt

ext4 write log file

/Data & Preprocessing & Plotting/iozone_log_ext4.txt

nilfs2 write log file

/Data & Preprocessing & Plotting/iozone_log_edited_nilfs2.txt