2019년 2학기 시스템 소프트웨어 1차 과제

-Log Structured File System Profiling –

제출자 (조장)2017320168 컴퓨터학과 김민수 → LKM 코드 작성 및 보고서 작성

(조원)2017320192 컴퓨터학과 최원미 → 커널 코드 수정 및 보고서 작성

제출일 2019.11.3 (블랙보드 기준)

주제 LFS(Log-structured File System) & Ext4 File System Profiling

환경 가상머신: Oracle VM virtualBox 6.0.14

운영체제: Ubuntu 16.04 LTS (64bit)

커널: Linux-4.4.0

파일시스템: Nilfs2, Ext4

사용 언어: C언어

코드 작성 툴: vim / Visual Studio Code 1.39.2

I/O 벤치마크 툴: iozone

하드웨어 스펙: AMD Ryzen 5 3600 6-Core Processor 3.59GHz 중 4개의 CPU

16GB 중 8GB의 RAM

256GB 중 40GB의 동적할당 디스크

Freeday 5일 중 3일 사용 (11/1~11/3)

1. 배경지식

1) Virtual File System (VFS)

VFS는 실제 파일 시스템의 구현과 사용자 프로세스 사이에 존재하는 추상화 계층에 존재하며, 파일 시스템 인터페이스를 유저 공간 프로그램에 제공하기 위해 구현된 커널의 서브 시스템이다. 또한 VFS는 C언어로 구현되어 있지만 객체 지향적 방식을 사용하여 시스템 콜을 호출하면 함수 포인터를 이용해 해당 파일 시스템이나 물리 매체의 종류에 맞게 overloading을 해준다. 이를 위해 VFS는 크게 superblock, inode, file, dentry 등 네 가지 데이터 구조체를 가지고 있다. Superblock은 마운트 시킨 파일 시스템의 정보를 저장하고 있다. Struct super_block으로 구현되어 있으며 내부에는 파일 시스템의 타입 s_type과 superblock 함수 포인터의 모음 s_op, inode 리스트 s_inodes 등의 정보를 가지고 있다. Inode는 파일 관리에 필요한 소유자 그룹, 접근 모드, 파일 형태, inode 번호 등의 정보를 저장하고 있다. 하나의 파일에 대해 하나의 inode를 생성할 수 있으며 각각의 고유한 inode 번호를 통해 파일을 식별 가능하다. Dentry는 디렉토리의 inode와 file path를 가지고 있는 객체로서 디스크에 저장되어 있지 않고 메모리에 저장되어 있는 In-memory data structure이다. 따라서 directory 접근을 효율적으로 하기 위하여 거의 모든 directory 정보를 dentry에 캐싱 하여 사용한다. 마지막으로 file은 프로세스가 open한 파일을 표현하는데 사용하는 구조체이며 이는 dentry 와 같이 파일시스템마다 가지는 구조체가 아니라 하나만 존재하는 In-memory data structure이다.

2) Log-structured File System (LFS)

LFS는 기존의 파일 시스템인 FFS(Fast File System)과 달리, 데이터 write시에 디스크에 sequential하게 쓰는 파일 시스템이다. 이렇게 하는 이유는 기존 파일 시스템에 있다. LFS가 고안되기 전, 기존 파일 시스템에서는 다음과 같은 문제점이 존재했다.

① 기존의 파일 시스템의 read 성능은 memory cache를 사용함으로써 성능이

많이 개선되었다. 반면에 write를 하는 경우 filesystem의 metadata가 여러 개로 나누어져 있어서, 여러 개의 small write를 수행해야 하는데, 이때 data update는 asynchronous하게 이루어 지지만 metadata update는 synchronous하게 이루어 진다. 따라서 여러 번의 metadata update를 수행하면서 seek time이 증가하기 때문에 disk의 성능은 낮아질 수밖에 없게 된다.

② Disk가 한번 fail되면 consistency를 체크하기 위해서 전체 disk를 스캔해야 한다. 그 이유는 위에서도 언급된 듯이 metadata update는 synchronous하게 되지만 data update가 asynchronous하게 됨으로 디스크가 커짐에 따라 복구하는 시간이 엄청 오래 걸린다.

이런 문제점들을 보완하기 LFS은 disk write를 한번에 모아서 처리하고 모든 새로운 정보를 append하는 방식으로 seek time을 없앴다. 또한 sequential structure를 적용하여 data를 처음부터 sequential하게 처리하기 때문에, disk를 복구하려고 할 때는 파일의 제일 뒤에 있는 정보만 체크하면 되게끔 하였다. 하지만 데이터를 sequential하게 쓴다는 특징 때문에 LFS는 다음과 같은 두 가지를 기능을 필요로 하였다.

① Read할 때 가장 최신의 Inode를 쉽게 찾기 위해 LFS는 디스크에 고정된 위치에 inode map이라는 구조체를 도입해서 가장 최신의 inode 위치를 저장하도록 하였다.

② Free space를 가능하게 크게 유지하기 위해 LFS는 디스크를 segment로 나눠서 쓴다. 그리고 쓰기 전에 몇 개의 segment를 메모리로 읽어서 live data는 다른 segment로 복사되고 나머지 data는 flushing을 한다.

하지만 이런 기능이 있음에도 불구하고 Free space management, Segment cleaning 등의 작업의 성능이 좋지 않아서 사용되지 않다가, 현재는 Flash Memory의 특성과 잘 맞아서 Flash memory를 위한 파일 시스템으로 사용한다.

3) Nilfs2

Nilfs2는 linux에서 지원하는 LFS이다. LFS의 특성에 맞게 crash로부터 복구하는 것

이 비교적 수월하고, 스냅샷을 통해 백업을 만들어 내기에 편하다. Nilfs2는, 한 번 쓴 곳은 지우지 않으면 다시 쓸 수 없는 flash memory의 특성과 잘 맞아서 현재는 SSD를 위한 파일 시스템으로 사용되기도 한다. Nilfs2의 큰 장점은 스냅샷을 계속해서 저장해서 원하는 시점의 파일 시스템으로 돌아갈 수 있다는 점이 있다. 이것이 가능한 이유는 한 번 쓰여진 데이터는 garbage collection을 하지 않는 이상 계속 유지되기 때문이다. 이렇게 생성된 checkpoint를 통해, disk fail이 발생해도 가장 최근의 checkpoint로 돌아가면 되기 때문에 복원이 쉽다는 장점이 있다.

4) bio 구조체, submit_bio 함수, nilfs_segment_buffer 구조체

① **bio**

bio 구조체는 block I/O 수행에 필요한 정보를 가지는 구조체이다. 각 bio는 저장영역에 포함된 sector 번호와 sector의 수, 입출력 연산에 연관된 메모리 영역을 기술하는 한 개 이상의 세그먼트를 포함한다. Bio들은 link 형태로 연결되어 있어 순차적으로 연산 된다.

② **submit_bio**

submit_bio는 block I/O 수행의 시작 지점이다. bio 구조체를 인자로 받아서, 이를 기반으로 block I/O 관련 여러 작업을 처리한다.

③ **struct nilfs_segment_buffer**

Nilfs2는 block I/O를 수행할 때, submit_bio 함수가 호출되기 전까지 자신만의 구조체인 nilfs_segment_buffer 구조체를 사용한다. 이 구조체에서는 nilfs2의 superblock의 정보를 가지고 있다. 이번 과제에서는 nilfs2 파일 시스템을 사용하기 위해 nilfs_segment_buffer 구조체 포인터 타입을 갖는 segbuf 변수를 통해 superblock에 대한 처리를 해주었다.

5) Loadable Kernel Module (LKM)

LKM은 동적으로 kernel의 기능(모듈)을 넣었다 뺐다 할 수 있는 모듈이다. 기본적

으로 Monolithic-kernel에서 Micro-kernel의 장점을 취하고 있다고 할 수 있다. LKM의 장점으로는 커널 기능의 확장이 용이하고 커널 주소 공간에서 수행되므로 성능 저하가 없다는 것이 있고, 단점으로는 잘못 작성된 LKM이 전체 시스템에 영향을 미칠 가능성이 있다는 것이다. LKM 파일은 .ko 의 확장자를 가지며, 보통 Makefile과 .c의 소스코드를 통해 얻을 수 있다. 또한 LKM을 커널에 load하기 위해선, 커널과 LKM의 버전이 일치해야 한다.

6) Proc File System

Proc file system은 유저 영역에서 커널 영역의 자료구조에 접근할 때에 사용되는 메모리에만 존재하는 가상 파일 시스템이다. Proc file system의 경우는 실제로 block device에 연결된 것이 아닌, 메모리 상에 mapping이 되어있기 때문에 물리적인 device를 필요로 하지 않는다. Printk의 경우 버퍼 사이즈가 제한되어 있고, 데이터의 가공 및 종합이 어렵다는 점에서 단점이 뚜렷하다. 하지만 proc file system의 경우는 runtime에 필요한 작업들을 코드를 통해 해줄 수 있다는 점에서 printk에 비해 장점이 많다.

2. 작성한 코드 설명

작성한 코드는 커널에서 수정한 코드를 먼저 설명한 후에 LKM 코드를 설명할 것이다.

1) block/blk-core.c

아래는 circular queue를 구현한 코드이다.

```
/*
  author: wonmi
*/
/* kernel modified part - start */
struct Info_node{
  unsigned long long block_no; // 1. block number
  struct timeval time; // 2. write time
  const char* name; // 3. file system name
};
struct Info_node Qnode[800];  // circular queue
int cnt = 0;  // circular queue current value index
struct timeval curtime; // to get write time
/* kernel modified part - end */
```

Figure 1. 커널 내에 구현한 circular queue 코드

먼저, 서로 다른 타입의 데이터들(block number, write time, file system 이름)을 저장하기 위해 circular queue의 하나의 node를 표현하는 struct Info_node를 선언하였다. 각 변수의 타입은 저장하려는 값들의 타입과 같게 설정하였다. Qnode[800]으로 크기 800의 circular queue를 만들었고, cnt를 통해 현재 데이터를 저장할 곳의 인덱스를 표현하였다. 마지막으로 write time을 저장하기 위한 변수인 curtime을 선언하였다.

아래는 ext4를 위한 submit_bio 함수 코드이다.

```
blk_qc_t submit_bio(int rw, struct bio *bio)
{
  bio->bi_rw |= rw;

  /*
   * If it's a regular read/write or a barrier with data attached,
   * go through the normal accounting stuff before submission.
   */
  if (bio_has_data(bio)) {
    unsigned int count;

    if (unlikely(rw & REQ_WRITE_SAME))
      count = bdev_logical_block_size(bio->bi_bdev) >> 9;
    else
      count = bio_sectors(bio);

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

      /* kernel modified part - start */
      if (bio != NULL) {
        do_gettimeofday(&curtime);  // get write time
        Qnode[cnt].block_no = bio->bi_iter.bi_sector; // block number
        if (bio->bi_bdev->bd_super != NULL) {
          Qnode[cnt].name = bio->bi_bdev->bd_super->s_type->name; // f
        }
        Qnode[cnt].time = curtime;  // time
        cnt = (cnt + 1) % 800;  // circular queue indexs
        /* kernel modified part - end */
      }
    } else {
      task_io_account_read(bio->bi_iter.bi_size);
      count_vm_events(PGPGIN, count);
    }

    if (unlikely(block_dump)) {
      char b[BDEVNAME_SIZE];
      printk(KERN_DEBUG "%s(%d): %s block %Lu on %s (%u sectors)\n",
      current->comm, task_pid_nr(current),
        (rw & WRITE) ? "WRITE" : "READ",
        (unsigned long long)bio->bi_iter.bi_sector,
        bdevname(bio->bi_bdev, b),
        count);
    }
  }

  return generic_make_request(bio);
}
```

Figure 2. submit_bio.c 함수 코드

submit_bio 함수에는 write를 실행할 때 사용 중인 file system, 해당 block number 및 시간 값을 메모리(circular queue)에 저장하기 위한 코드를 추가하였다. 이번 프로젝트에서는 write하는 경우에만 정보를 저장하면 되므로, 모든 코드는 write하는 경우의 branch에서 작성하였다(rw & write).

먼저 do_gettimeofday(&curtime)은, 디바이스 드라이버 상에서 시간을 얻어 오기 위한 함수이고, 그 시간을 curtime 변수에 저장한다. 수정한 코드의 if문의 조건은 모두 null pointer exception을 피하기 위한 조건문이다. bio 구조체 포인터에서 bi_iter.bi_sector를 통해 block number를 가져왔고, bio 구조체 포인터 안의 bi_bdev 구조체 포인터 안의 bd_super구조체 포인터 안의 bd_super구조체 포인터 안의 s_type구조체 포인터 안의 name을 통해 현재 파일 시스템의 이름을 가져왔다. 마지막으로 크기가 800인 circular queue를 구현하기 위해, 인덱스를 조정하는 코드인 cnt = (cnt+1) % 800 도 추가하였다.

아래는 LKM에서 circular queue에 접근하기 위해 사용하는 EXPORT_SYMBOL을 작성한 부분이다.

```
EXPORT_SYMBOL(submit_bio);
/* kernel modified part - start */
EXPORT_SYMBOL(Qnode);
EXPORT_SYMBOL(cnt);
/* kernel modified part - end */
```

Figure 3. circular queue들에 대한 EXPORT_SYMBOL들

EXPORT_SYMBOL을 통해 circular queue와 그에 대한 인덱스를 LKM에서 접근할 수 있게끔 하였다. LKM 코드에서는 export 된 symbol을 사용하기 위해 extern 예약어를 사용하였다.

다음은 nilfs2를 위한 blk-core.c의 submit_bio 함수이다.

```
blk_qc_t submit_bio(int rw, struct bio *bio)
{
  bio->bi_rw |= rw;

  /*
   * If it's a regular read/write or a barrier with data attached,
   * go through the normal accounting stuff before submission.
   */
  if (bio_has_data(bio)) {
    unsigned int count;

    if (unlikely(rw & REQ_WRITE_SAME))
      count = bdev_logical_block_size(bio->bi_bdev) >> 9;
    else
      count = bio_sectors(bio);

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

      /* kernel modified part - start */
      if (bio != NULL) {
        if (bio->bi_bdev->bd_super != NULL && strcmp(bio->bi_bdev->bd_super->s_type->name, "nilfs2") == 0) {
          do_gettimeofday(&curtime);  // get write time
          Qnode[cnt].block_no = bio->bi_iter.bi_sector; // block number
          Qnode[cnt].name = bio->bi_bdev->bd_super->s_type->name; // file system name
          Qnode[cnt].time = curtime;  // time
          cnt = (cnt + 1) % 800;  // circular queue indexs
        }
        /* kernel modified part - end */
      }
    } else {
      task_io_account_read(bio->bi_iter.bi_size);
      count_vm_events(PGPGIN, count);
    }

    if (unlikely(block_dump)) {
      char b[BDEVNAME_SIZE];
      printk(KERN_DEBUG "%s(%d): %s block %Lu on %s (%u sectors)\n",
      current->comm, task_pid_nr(current),
        (rw & WRITE) ? "WRITE" : "READ",
        (unsigned long long)bio->bi_iter.bi_sector,
        bdevname(bio->bi_bdev, b),
        count);
    }
  }

  return generic_make_request(bio);
}
```

Figure 4. nilfs2를 위한 submit_bio 코드

Figure 4가 Ext4를 위한 submit_bio 함수와 다른 점은, 이 코드는 I/O를 실행하는 파일 시스템이 nilfs2일 경우에만 데이터를 저장하게 했다는 것이다. 이것 외에 다른 점은 없다.

2) Makefile

아래는 proc을 위한 LKM 코드를 컴파일하기 위해 사용한 Makefile 코드이다.

```
obj-m += proc.o

KDIR = /usr/src/linux-4.4

all:
	$(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules
clean:
	rm -rf *.o *.ko *.mod.* *.symvers *.order
```

Figure 5. Makefile 코드

Makefile의 obj-m에는 컴파일 할 LKM 코드의 object 파일을 넣어주었다. 작성한 코드의 이름이 proc.c로 했으므로, 그의 object 파일인 proc.o를 추가해주었다. KDIR에는 사용하는 커널의 디렉토리의 경로를 넣었다. all은 make 명령어를 실행한 경우에 실행되는 명령어고, PWD는 현재 디렉토리의 경로를 의미한다. clean은 make clean 실행 시에 현재 디렉토리에서 삭제할 파일들에 대해서 적어주었다.

3) proc.c (LKM 코드)

```
/**
 *  author: minsu kim
 */
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/proc_fs.h>
#include <linux/time.h>
#include <asm/uaccess.h>

#define PROC_DIRNAME    "proj1"             // proc file system directory name
#define PROC_FILENAME   "proj1file"         // proc file system file name
#define Q_SIZE          800                 // circular queue size

static struct proc_dir_entry *proc_dir;     // proc file system directory
static struct proc_dir_entry *proc_file;    // proc file system file

// struct Info_node is one node for circular queue
// same as kernel's Info_node struct
struct Info_node {
    unsigned long long block_no;    // 1. block number
    struct timeval time;            // 2. write time
    const char* name;               // 3. file system name
};
extern struct Info_node Qnode[Q_SIZE];  // get kernel-defined circular queue
extern int cnt;                         // index for circular queue's current value

struct tm   times;              // struct tm is for formatting time data in Info_node variable
char        info[800][100];     // info is temporary variable for connecting kernel-user memory
```

Figure 6. LKM 코드의 변수 및 매크로들 선언부

먼저, PROC_DIRNAME과 PROC_FILENAME으로 proc 파일 시스템에서 사용할 proc 파일의 디렉토리 이름과 파일 이름을 설정해주었다. Q_SIZE는 커널 코드에서 사용하던 circular queue의 크기이고, 커널에서 설정한 값과 같이 800이다. proc_dir과 proc_file은 아래에서 proc 파일 시스템에 디렉토리 및 파일을 생성하기 위해 사용하는 포인터 변수이다. Info_node 구조체는 커널에서 수정한 코드와 같은, circular queue의 한 node를 나타내는 구조체이다. 여기서 가장 중요한 것은 extern을 사용한 부분인데, extern은 커널에서 EXPORT_SYMBOL을 통해 내보내 준 변수들을 사용할 수 있게 해주는 예약어이다. extern을 통해 가져온 변수는 Qnode와 cnt이고, 이는 각각 커널에서 수정한 코드와 같이 circular queue, 인덱스를 의미한다. times는 circular queue에 저장된 시간 값을 formatting하는 데에 필요한 변수이다. info 변수도 마찬가지로 user buffer에 kernel에서의 값을 넣어주기 위한 임시 변수이다.

```c
// custom open function for open proc file
static int my_open(struct inode *inode, struct file *file) {
    printk(KERN_INFO "Proc Module Open!!\n");

    return 0;
}

// custom read function for reading proc file
static ssize_t my_read(struct file *file, char *buffer, size_t length, loff_t *offset) {
    // copy kernel's data into user buffer
    if ( copy_to_user(buffer, info, sizeof(info)) ) {   // copy_to_user returns 0 in error
      return -EFAULT;
    }
    return length;
}

// custom write function for writing to proc file
static ssize_t my_write(struct file *file, const char __user *user_buffer, size_t count, loff_t *ppos) {

    int     i = 0;

    // write data into queue
    for (i=cnt+1; i<Q_SIZE; ++i) {
        time_to_tm(Qnode[i].time.tv_sec, 0, &times);
        sprintf(info[i], "File System: %s, Block Number: %lld, Time: %d:%d:%d:%ld\n",
          Qnode[i].name, Qnode[i].block_no, times.tm_hour, times.tm_min, times.tm_sec, Qnode[i].time.tv_usec);
    }
    for (i=0; i<=cnt; ++i) {
        time_to_tm(Qnode[i].time.tv_sec, 0, &times);
        sprintf(info[i], "File System: %s, Block Number: %lld, Time: %d:%d:%d:%ld\n",
          Qnode[i].name, Qnode[i].block_no, times.tm_hour, times.tm_min, times.tm_sec, Qnode[i].time.tv_usec);
    }

    return count;
}
```

Figure 7. proc 파일을 open/read/write할 때에 실행시킬 custom 함수들

my_open 함수는 proc file이 open되었을 때 실행되는 함수이다. printk 함수는 proc 파일이 정상적으로 open 되었는지를 보기 위해 작성하였다. my_read 함수는 proc file을 read하는 경우에 호출되는 함수이다. 이 함수는 circular queue에 저장된 정보들을 user buffer로 복사해주는 역할을 한다. 유저 영역에서는 커널 메모리의 주소에 직접적으로 접근할 수가 없어서 커널 메모리의 데이터들을 user buffer로 옮기는 작업이 필요한데, 이는 asm/uaccess.h에 정의된 copy_to_user 함수를 통해 구현하였다. 마지막으로 my_write 함수는 proc file에 write를 할 경우에 호출되는 함수이다. 이 함수는 임시 버퍼 info에 block number, write time, 파일 시스템 이름의 총 3개의 데이터를 저장하는 역할을 하는 함수이다.

아래는 proc.c 코드에서, LKM이 정상적으로 load 되었는지 확인하는 코드이다.

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

    // make directory in proc file system
    proc_dir = proc_mkdir(PROC_DIRNAME, NULL);
    // create proc file in proc file system
    proc_file = proc_create(PROC_FILENAME, 0600, proc_dir, &myproc_fops);

    return 0;
}

static void __exit simple_exit(void) {
    printk(KERN_INFO "Simple Module Exit!!\n");

    return ;
}

module_init(simple_init);
module_exit(simple_exit);

MODULE_AUTHOR("2017320168 2017320192");
MODULE_DESCRIPTION("Project1 LKM for profiling file systems");
MODULE_LICENSE("GPL");
MODULE_VERSION("NEW");
```

Figure 8. proc.c의 LKM 동작 확인 코드

Figure 7에서, simple_init 함수는 LKM이 insmod 명령어를 통해 실행되었을 경우에 호출되는 함수이다. LKM이 정상적으로 load되는지 확인하기 위해 printk 함수를 넣었고, proc 파일 시스템에서 사용할 디렉토리, proc 파일을 생성하는 코드도 추가

하였다. simple_exit는 rmmod 명령어에 의해 LKM이 종료될 때 호출되는 함수이다. LKM이 정상적으로 unload 되는지 확인하기 위해 printk를 사용해서 커널에 메세지를 출력했다.

4) segbuf.c

아래는 segbuf.c의 nilfs_segbuf_submit_bio 함수의 코드이다.

```
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;

  if (segbuf->sb_nbio > 0 &&
      bdi_write_congested(segbuf->sb_super->s_bdi)) {
    wait_for_completion(&segbuf->sb_bio_event);
    segbuf->sb_nbio--;
    if (unlikely(atomic_read(&segbuf->sb_err))) {
      bio_put(bio);
      err = -EIO;
      goto failed;
    }
  }

  bio->bi_end_io = nilfs_end_bio_write;
  bio->bi_private = segbuf;
  /*
    author: wonmi
  */
  /* kernel modified part - start */
  bio->bi_bdev->bd_super = segbuf->sb_super;  // change to nilfs2
  /* kernel modified part - end */
  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);
  wi->start = wi->end;
  return 0;

 failed:
  wi->bio = NULL;
  return err;
}
```

Figure 9. nilfs_segbuf_submit_bio 함수

이 함수에 추가한 코드는, nilfs2를 사용하는 경우 nilfs2의 superblock을 할당하는 코드이다. 이를 통해 마운트 된 nilfs2 파일 시스템을 사용할 수 있게 된다.

3. 실행 방법에 대한 간략한 설명

0) Ext4용 커널 코드 컴파일 및 재부팅
   - 제출된 파일의 ext4 폴더 내의 blk-core.c로 커널 코드를 대체해서 컴파일 후 재부팅
1) LKM 코드 컴파일
   - 작성한 코드를 Makefile을 통해 컴파일

     (작성한 LKM 코드가 있는 디렉토리에서) make
   - 결과 파일 중 proc.ko가 생성될 것(코드 파일 이름을 proc.c로 함)
2) LKM load하기
   - (proc.ko가 있는 디렉토리에서) sudo insmod proc.ko
   - Load된 모듈은 lsmod 명령어를 통해 확인 가능
   - 모듈이 load되면, proc 파일 시스템에 proj1 디렉토리와, 그 안에 proj1file이 생성 (LKM 코드에서 이름을 이렇게 설정함)
3) iozone으로 ext4 테스트
   - ~/Desktop/iozone3_414/src/current/iozone -i 0 -f ~/Desktop/iotest/anyfile
   - iozone으로 write를 해서, 커널 내에 구현한 circular queue를 채움
4) echo 명령어를 통해 proc의 write 함수 호출
   - sudo sh -c "echo "ext4" >> /proc/proj1/proj1file"
5) cat 명령어를 통해 데이터 읽기 및 읽은 데이터 저장
   - sudo sh -c "cat /proc/proj1/proj1file >> ext4log.txt"
6) nilfs2용 커널 코드 컴파일 및 재부팅 후 nilfs2 마운트(다운로드 및 설정은 생략)
   - 제출된 파일의 nilfs2 폴더내에 있는 blk-core.c로 커널 코드를 대체한 후 컴파일 후 재부팅

- nilfs2를 위한 blk-core.c 코드가 없으면 로그가 ext4와 섞여서 나오기 때문에 따로 진행함
- 원하는 디렉토리에서 가상디스크 생성 후 원하는 디렉토리에 nilfs2 마운트
- dd if=/dev/zero of=./diskfile bs=1024 count=2000000 → 가상 디스크 생성(2GB)
- mkfs.nilfs2 ./diskfile → 디스크를 nilfs2로 포멧
- sudo losetup /dev/loop0 ./diskfile → disk를 I/O device로 등록
- mkdir nil → 마운트 할 디렉토리 생성
- sudo mount -t nilfs2 /dev/loop0 nil → 마운트

7) iozone으로 nilfs2 테스트
   - sudo sh -c "~/Desktop/iozone3_414/src/current/iozone -i 0 -f ~/Desktop/iotest/nil/anyfile"
8) echo 명령어를 통해 proc의 write 함수 호출
   - sudo sh -c "echo "nilfs" >> /proc/proj1/proj1file"
9) cat 명령어를 통해 데이터 읽기 및 읽은 데이터 저장
   - sudo sh -c "cat /proc/proj1/proj1file >> nilfs2log.txt"
10) 로그들 모두 800개씩만 얻어 냄
   - head -n 800 ext4log.txt >> extlog.txt
   - head -n 800 nilfs2log.txt >> nillog.txt
11) 실행 결과(로그 파일) 예시

```
File System: ext4, Block Number: 4204512, Time: 18:53:49:612040
File System: ext4, Block Number: 4468408, Time: 18:53:49:612041
File System: ext4, Block Number: 4637520, Time: 18:53:49:612042
File System: ext4, Block Number: 20971528, Time: 18:53:49:612043
File System: ext4, Block Number: 20971552, Time: 18:53:49:612044
File System: ext4, Block Number: 20971568, Time: 18:53:49:612307
File System: ext4, Block Number: 20971624, Time: 18:53:49:612308
File System: ext4, Block Number: 20982688, Time: 18:53:49:612310
File System: ext4, Block Number: 20982720, Time: 18:53:49:612311
File System: ext4, Block Number: 20982856, Time: 18:53:49:612312
File System: ext4, Block Number: 20982864, Time: 18:53:49:612312
File System: ext4, Block Number: 20982872, Time: 18:53:49:612313
File System: ext4, Block Number: 20984888, Time: 18:53:49:612314
File System: ext4, Block Number: 41943040, Time: 18:53:49:612316
File System: ext4, Block Number: 41943048, Time: 18:53:49:612317
File System: ext4, Block Number: 41943192, Time: 18:53:49:612318
File System: ext4, Block Number: 41943296, Time: 18:53:49:612319
File System: ext4, Block Number: 41955560, Time: 18:53:49:612320
File System: ext4, Block Number: 42008712, Time: 18:53:49:612320
File System: ext4, Block Number: 50331672, Time: 18:53:49:612321
File System: ext4, Block Number: 58720256, Time: 18:53:49:612323
File System: ext4, Block Number: 58720352, Time: 18:53:49:612325
File System: ext4, Block Number: 58720408, Time: 18:53:49:612326
File System: ext4, Block Number: 58720528, Time: 18:53:49:612326
```

Figure 10. ext4 파일시스템 로그 예시

```
File System: nilfs2, Block Number: 526160, Time: 18:40:41:794372
File System: nilfs2, Block Number: 0, Time: 18:40:41:794535
File System: nilfs2, Block Number: 526288, Time: 18:40:41:799855
File System: nilfs2, Block Number: 527448, Time: 18:40:41:803919
File System: nilfs2, Block Number: 528600, Time: 18:40:42:26017
File System: nilfs2, Block Number: 528736, Time: 18:40:42:31258
File System: nilfs2, Block Number: 529896, Time: 18:40:42:35609
File System: nilfs2, Block Number: 531040, Time: 18:40:42:249787
File System: nilfs2, Block Number: 531168, Time: 18:40:42:255035
File System: nilfs2, Block Number: 532328, Time: 18:40:42:258475
File System: nilfs2, Block Number: 533472, Time: 18:40:42:417970
File System: nilfs2, Block Number: 533600, Time: 18:40:42:429878
File System: nilfs2, Block Number: 534760, Time: 18:40:42:433992
File System: nilfs2, Block Number: 535904, Time: 18:40:42:593394
File System: nilfs2, Block Number: 536032, Time: 18:40:42:601061
File System: nilfs2, Block Number: 537192, Time: 18:40:42:604955
File System: nilfs2, Block Number: 538336, Time: 18:40:47:615673
File System: nilfs2, Block Number: 538464, Time: 18:41:0:540907
File System: nilfs2, Block Number: 539624, Time: 18:41:0:543922
File System: nilfs2, Block Number: 540672, Time: 18:41:0:543935
File System: nilfs2, Block Number: 540776, Time: 18:41:0:987681
File System: nilfs2, Block Number: 3999992, Time: 18:41:0:987755
File System: nilfs2, Block Number: 540912, Time: 18:41:0:993119
File System: nilfs2, Block Number: 542080, Time: 18:41:0:996308
```

Figure 11. nilfs2 파일시스템 로그 예시

4. 결과 그래프 및 설명

아래는 excel으로 그린 두 파일 시스템의 로그에 대한 그래프이다. x축은 시간, y축은 block number이다.

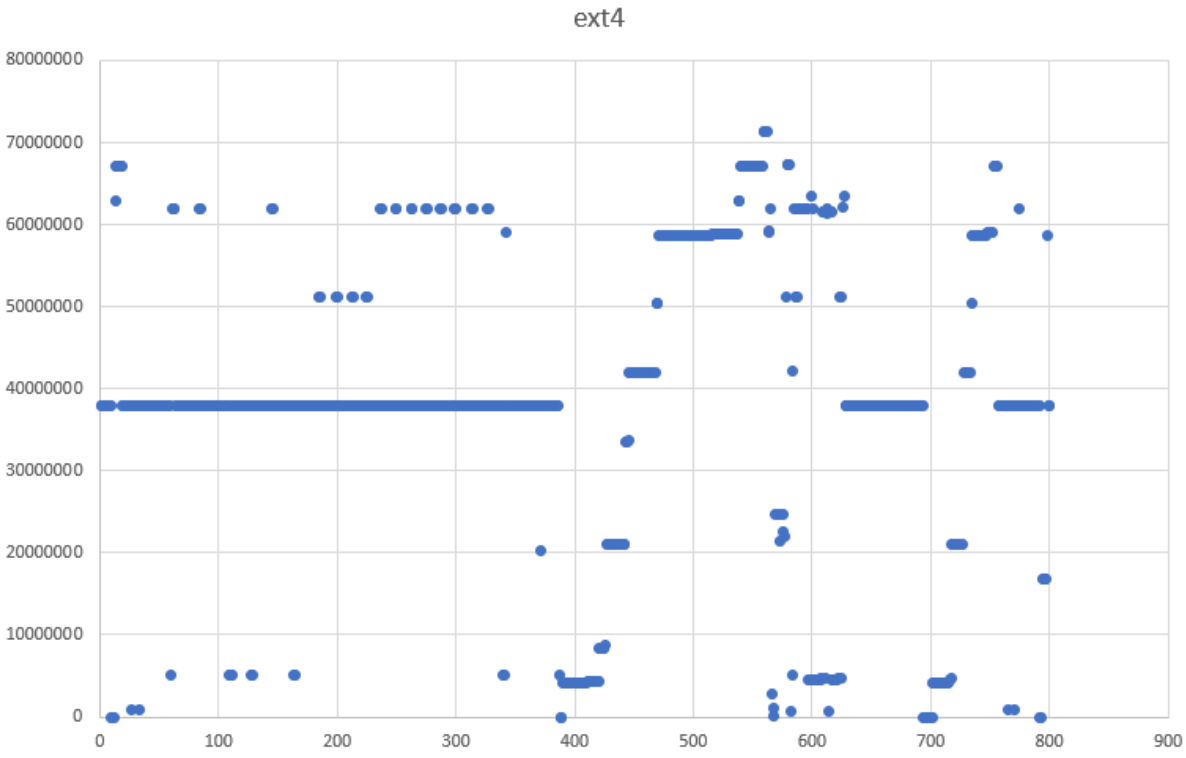

Figure 12. ext4 파일 시스템 그래프

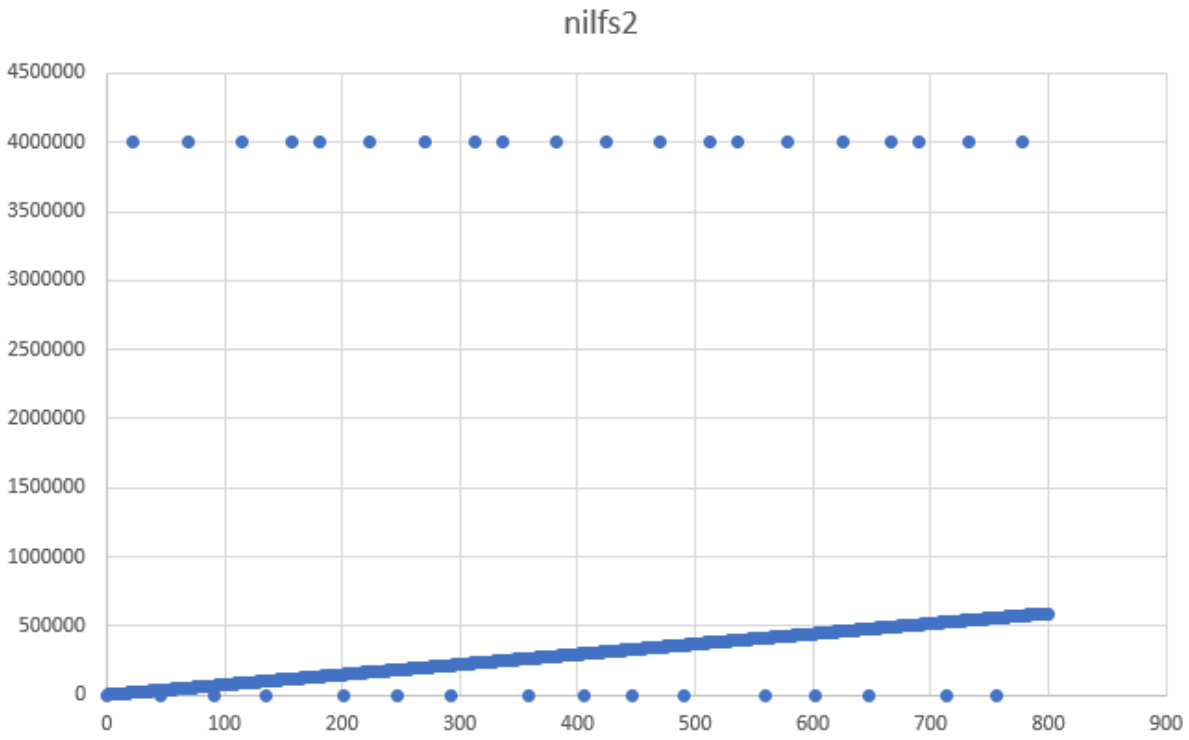

Figure 13. nilfs2 파일 시스템 그래프

먼저 두 그래프의 가장 큰 차이점은, ext4의 경우 write 하는 block의 번호가 일정하지 않은 반면 nilfs2의 경우 write 하는 block의 번호가 0번부터 sequential하게 형성되어 있다. Ext4의 경우 새로운 데이터를 write할 때, 같은 실린더 내에서 data를 쓴 후 inode의 위치에 데이터를 다시 쓴다. 그러므로 접근하는 disk block의 번호에 변동이 많을 수밖에 없다. Nilfs2는 LFS의 한 종류이므로, 처음부터 sequential하게 쓰게 되므로, 그래프에서 나타나는 경향은 당연하다고 볼 수 있다. 여기서 주목할 점은, nilfs2가 0번과 3999992번에 주기적으로 write를 하고 있다는 점인데, 0은 current inode map의 주소 값을 저장하는 checkpoint region이라고 알고 있지만 3999992는 어떤 의미를 갖는지 알 수 없었다. 하지만 3999992도 0과 같은 어떤 fixed position이고, fixed position에는 superblock과 checkpoint region이 위치하므로, 3999992에도 둘 중 하나가 저장될 것이라고 유추할 수 있다.

5. 과제 수행 시 어려웠던 부분과 해결 방법

1) 과제 수행에 필요한 커널 컴파일

과제에서 커널을 컴파일 하거나 다른 버전의 커널로 부팅하거나 menuconfig 작업 등을 할 때, 과제의 실습자료와는 다른 결과를 도출하는 경우가 있었다. 가령 menuconfig를 할 때 오류가 발생해서 구글에서 해답을 찾아서 진행하거나, nilfs2를 enable할 때의 설정 메뉴 예시는 위 환경에서 진행한 것과는 달랐다. 하지만 필요한 부분을 잘 찾거나, 구글에 검색하는 등의 방법으로 비교적 쉽게 해결할 수 있었다.

2) proc file system에서 로그 출력하기

과제에서 proc file system을 통해서 파일 시스템들에 대한 profiling 로그를 출력해야 했는데, 1학기의 운영체제 강의에서는 printk와 dmesg로 쉽게 출력이 가능했어서 어려움이 없었지만, 버퍼의 개념을 이해하는 데에 시간이 걸렸다. 하지만 과제 실습 자료와 페이스북 그룹에서의 질문에 대한 답변을 통해 copy_to_user 함수를 사용해야 한다는 것을 알았고, 조원과 함께 고민한 결과 커널 메모리의 값을 user buffer로 넘겨주는 방법을 알아내서 출력에 성공하였다.

3) Nilfs2 profiling의 결과

과제를 진행하는 동안은 Nilfs2의 경우 특정 디렉토리에서만 사용하는 파일 시스템이기 때문에, 커널 로그가 ext4와 함께 찍혀서 나왔었다. 처음에는 이렇게 되는 이유를 이해하지 못해서 구현을 잘못한 것으로 생각해서 코드를 많이 수정했었는데, 조원과 같이 고민해본 결과 ext4에 대한 로그도 같이 출력되는 것이 정상이라는 것을 깨닫고 nlifs2를 위한 커널 코드를 만들어서 nilfs2에 대한 로그만 출력하게 하는 것으로 해결했다.

4) Nilfs2 마운트 관련 이슈

처음에는 리눅스 부팅 시마다 nilfs2를 마운트 해야 한다는 것을 몰라서 어려움을 겪었지만, mount | grep nilfs 명령어를 통해 매번 마운트를 다시 해야 한다는

것을 알 수 있었다.