Assignment 1

서로 다른 파일시스템의 쓰기 분석

2011210028 홍만기
2013210111 남세현

0. 제출상세

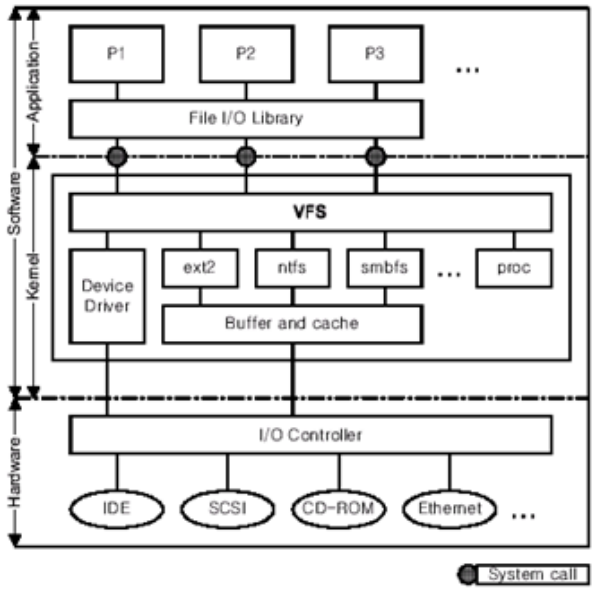
2011210028 홍만기 – 이론파트 조사, 보고서 편집
2013210111 남세현 – 상세 코드구현/정리, 자료분석
제출일자 : 2015 년 11 월 4 일 (Free day 5 일 + 1 일 사용)

1. 배경지식

A. Virtual File System (VFS)

VFS는 파일 시스템의 추상계층이다. 사용자가 여러 파일 시스템을 같은 방식으로 접근할 수 있게 만드는 것이 VFS의 목적이다. 이를 통해 사용자는 자신이 접근하는 파일시스템의 상세를 알지 못해도 무리없이 다수의 파일시스템에 접근하는 것이 가능하다.

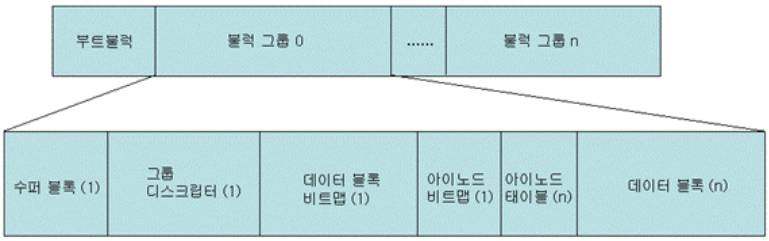
VFS는 커널과 파일시스템과의 interface를 정의한다. 정해진 규격에 맞춰서 기존 파일시스템들을 호환시키는 것이다. 이로써 파일시스템간의 상호동작 또한 가능해진다.



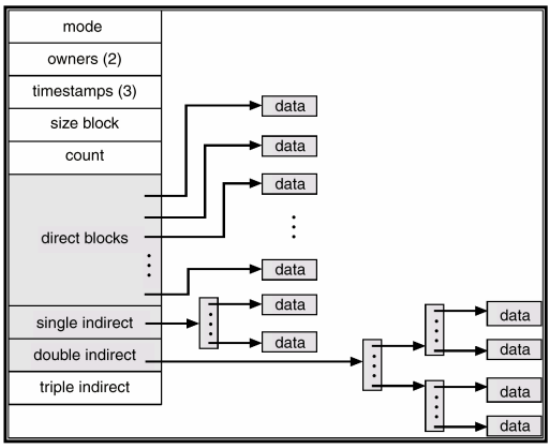
VFS 전체구조

VFS를 구현할 때에는 OOP의 overloading 개념을 적용한다. 시스템 호출시 해당 파일이 속한 파일시스템에 맞춰서 동작을 매핑해주는 방식이다. 하지만, OOP가 명시적으로 지원되지 않기 때문에 각 객체는 구조체의 형식을 따른다. 이 때 서로 다른 파일시스템간의 공통된 부분을 통합한 VFS 객체를 메모리 상에서 사용한다.

Superblock object </linux/fs.h> - 마운트된 파일 시스템 정보를 저장한다. 슈퍼블록의 자료 구조, 파일 시스템의 크기, 블록의 수, 이용가능 블록의 수, 빈 블록에서 다음 빈 블록까지의 index, inode목록의 크기 등이 저장된다.



Inode object </linux/fs.h> - Unix 상의 File Control Block. 특정 파일에 대한 일반 정보를 저장한다. 각 파일은 inode number를 가진다.

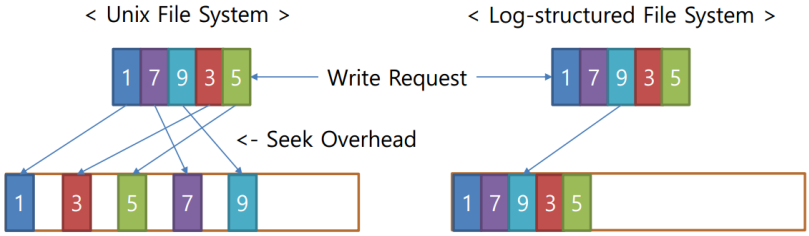


File object </linux/fs.h> - 열린 파일과 프로세스 사이의 상호 작용과 관련한 정보를 저장한다.

Dentry object </linux/dcache.h> 디렉토리 항목과 대응하는 파일간 연결에 대한 정보를 저장한다.

B. Log-Structured File System (LFS)

CPU와 메모리 성능의 증가로 캐싱되는 파일의 양이 많아짐에 따라 read명령이 그렇게 많은 I/O traffic을 발생하지 않게 되었다. 그로 인해 write request가 I/O traffic의 대부분을 차지하게 되었고, 이를 모아 Disk에 순차적으로 정리하여 저장하기로 한 것이 LFS이다.



LFS는 Unix 시스템과 같이 Inode를 사용하나, 효율적인 Inode 관리를 위해 Inode Map이 존재한다.

LFS가 효율적으로 동작하기 위해선 연속된 공간을 확보할 필요가 있다. 하지만 파일의 읽기와 쓰기를 반복하다보면 디스크에 순차적으로 저장되었던 데이터들 사이에 구멍이 생기는데, 이를 관리하는 것이 LFS의 큰 이슈이다. 주로 유효한 로그들은 한곳에 모으는 방식의 일종의 조각모음을 사용하여 연속된 공간을 확보한다.

2. 코드분석

A. 커널 수정 및 이미지 퓨징

Bio 구조체는 block I/O에 필요한 정보들을 담고 있다. 그래서 Block I/O가 시작된다면 submit_bio라는 함수에서부터 시작을 한다. 그리하여 어떤 블럭에서 write를 하는지 알기 위해서 submit_bio를 다음과 같이 수정하였다.

```
1643  void submit_bio(int rw, struct bio *bio)
1644  {
1645      int count = bio_sectors(bio);
1646
1647      bio->bi_rw |= rw;
1648
1649      /*
1650       * If it's a regular read/write or a barrier with data attached,
1651       * go through the normal accounting stuff before submission.
1652       */
1653      if (bio_has_data(bio) && !(rw & REQ_DISCARD)) {
1654          if (rw & WRITE) {
1655              count_vm_events(PGPGOUT, count);
1656          } else {
1657              task_io_account_read(bio->bi_size);
1658              count_vm_events(PGPGIN, count);
1659          }
1660
1661      if (unlikely(block_dump)) {
1662              struct file* filp;
1663              char logString[512];
1664              char b[BDEVNAME_SIZE];
1665              if(strcmp("iozone", current->comm) == 0)
1666              {
1667                  int length = sprintf(logString,  "%Lu\n",
1668                  (unsigned long long)bio->bi_sector);
1669                  filp = file_open("/proc/myproc/myproc", O_WRONLY, 0);
1670                  if(filp)
1671                  {
1672                      file_write(filp, filp->f_pos, logString, length);
1673                      file_close(filp);
1674                  }
1675                  else
1676                      printk("nowritten\n");
1677              }
1678          }
1679      }
1680
1681      generic_make_request(bio);
1682  }
1683  EXPORT_SYMBOL(submit_bio);
```

1661 : /proc/sys/vm/block_dump 의 값이 0이 아니면 내부 로직이 실행된다.
터미널에서echo 1 > /proc/sys/vm/bloc_dump 커맨드로 설정해줄 수 있다.
1665 : iozone에 의한 block I/O일 때만을 가려낸다. 수많은 프로세스가 I/O를 하고 있기 때문에, 모든 I/O에 대해서 덤핑을 하면 디바이스에 과부하가 걸리는 현상이 있기 때문이다.
1669 ~ 1676 : Kernel Module로 만들 ProcFS를 Open한다. 모듈이 정상적으로 작동하면 filp는 NULL이 아니게 되는데, 그런 경우 블럭 위치를 ProcFS에 찍는다.

위 코드에는 file_open, file_wrtie, file_close라는 사용자지정함수가 있다. 커널 내부에서는 sys_open, sys_write, sys_close를 사용해야 하는데 몇가지 문제가 있어서 sys_open 등을 직접

구현하여 사용하였다. 그것의 코드는 아래와 같다.

```
1610  struct file* file_open(const char* path, int flags, int rights) {
1611      struct file* filp = NULL;
1612      mm_segment_t oldfs;
1613      int err = 0;
1614
1615      oldfs = get_fs();
1616      set_fs(get_ds());
1617      filp = filp_open(path, flags, rights);
1618      set_fs(oldfs);
1619      if(IS_ERR(filp)) {
1620          err = PTR_ERR(filp);
1621          return NULL;
1622      }
1623      return filp;
1624  }
1625
1626  void file_close(struct file* file) {
1627      filp_close(file, NULL);
1628  }
1629
1630  int file_write(struct file* file, unsigned long long offset, unsigned char* data, unsigned int size) {
1631      mm_segment_t oldfs;
1632      int ret;
1633
1634      oldfs = get_fs();
1635      set_fs(get_ds());
1636
1637      ret = vfs_write(file, data, size, &offset);
1638
1639      set_fs(oldfs);
1640      return ret;
1641  }
```

출처 : http://stackoverflow.com/questions/1184274/how-to-read-write-files-within-a-linux-kernel-module

수정한 커널을 크로스 컴파일 한 후, 그 이미지를 디바이스에 업로드 한다.
과정은 tizen wiki에 나와있는 아래 자료에 맞게 진행되었다.

Kernel Build

Follow below steps to build the Tizen ARM kernel. (Tizen 2.0 - 2.2)

1. Install and setup cross compile tools on your system if the target and your host are different (e.g., x86).
2. You may use Linaro toolchain binaries or Ubuntu package of them and have your environment setup for the cross tools (e.g., export CROSS_COMPILE=....)
3. If your kernel source has been used to create binaries for other architecture, please start with cleaning them up.
   $ make distclean
4. Setup the .config file for RD-210
   $ make ARCH=arm trats_defconfig
5. Or for RD-PQ
   $ make ARCH=arm trats2_defconfig
6. Then, after reconfiguring per your needs (e.g., make ARCH=arm menuconfig) or using the stock configuration (no modifications), build it.
   $ make ARCH=arm uImage
7. Build and make kernel module image as well. Note that you may need to do sudo first to let sudo -n work in the script
   $ sudo ls
   $ scripts/mkmodimg.sh
8. Send the images to the target via lthor
   $ lthor arch/arm/boot/uImage usr/tmp-mod/modules.img
9. Or, you may make your own tar file from the two files
   $ tar cf FILENAME_YOU_WANT.tar -C arch/arm/boot uImage -C ../../../usr/tmp-mod modules.img

(출처 : https://wiki.tizen.org/wiki/Porting_Guide/Getting_Source_Code%26Build)

B. 모듈 프로그래밍

Custom ProcFS를 사용하기 위해서 Kernel Module을 프로그래밍하여야 한다.
일반적인 Kernel Moudle에서 init_module에 해당하는 함수에서 create_proc_entry 함수로 /proc/ directory 아래의 entry를 만들 수 있다.

그 후 만들어진 /proc /파일에 fop_s, 즉 File Operator들을 직접 정의를 하여 구현한다. 여기에서는 File Write, File Read를 수정하였다.

수정 방향은 다음과 같다.

- 메모리 상에 예상되는 벤치마크 로그량보다 충분히 큰 버퍼(512KB)를 생성.
- 버퍼에 얼마나 로그가 입력되었는지 길이를 저장하는 변수를 생성(Length).
- Block Dump 요청마다 해당 로그를 버퍼에 작성 후 Length 변수를 로그 길이만큼 증가.
- 출력(read) 요청시 요청되는 파일 포인터로부터 요청되는 count 만큼의 데이터를 copy_to_user 함수로 복사하여 리턴해줌.

```
1  #include <linux/module.h>
2  #include <linux/kernel.h>
3  #include <linux/init.h>
4  #include <linux/proc_fs.h>
5  #include <asm/uaccess.h>
6
7  #define min(a,b) \
8  ({ __typeof__ (a) _a = (a); \
9  __typeof__ (b) _b = (b); \
10 _a < _b ? _a : _b; })
11 #define PROC_DIRNAME "myproc"
12 #define PROC_FILENAME "myproc"
13 #define MAX_LOG_LENGTH (1024*512)
14 static struct proc_dir_entry *proc_dir;
15 static struct proc_dir_entry *proc_file;
16
17 static char stringBuffer[MAX_LOG_LENGTH];
18 static size_t bufferOffset;
19
20 static int my_open(struct inode *inode, struct file* file)
21 {
22     printk(KERN_INFO "module file open");
23     return 0;
24 }
25
26 static ssize_t my_write(struct file *file, const char __user *user_buffer, size_t count, loff_t *ppos)
27 {
28     if(bufferOffset + count > MAX_LOG_LENGTH)
29         return count;
30     if( copy_from_user(stringBuffer + bufferOffset, user_buffer, count) ) {
31         return -EFAULT;
32     }
33     bufferOffset += count;
34
35     return count;
```

11~12 : /proc 아래의 directory 이름과 entry file 이름에 해당하는 값
13 : 메모리 상에 존재하게 될 버퍼의 크기(512kb로, 충분한 크기)
14~15 : /proc 아래의 directory, entry file에 해당하는 file structure instance.
17 : 메모리상에 존재하게 될 버퍼.
18 : 버퍼에 얼마나 쓰였는지 나타내는 변수.
20 : /proc/PROC_DIRNAME/PROC_FILENAME을 open했은 경우 실행될 함수. “module file open” 라는 메시지를 dmesg를 통해 확인 가능하다.
26 : file write에 해당하는 함수. 리턴값은 정상적으로 처리된 데이터의 길이. 파라미터 *file은 접근한 파일 포인터, user_buffer은 write하고자 하는 데이터가 들어있는 버퍼, count는 user_buffer의 길이, 마지막 매개변수 ppos는 void* data로도 알려져 있는데, private data로 본 과제에서는 다루지 않는다.
28 : 버퍼 이상으로 데이터를 입력하려고 할 경우 버퍼에 작성은 하지 않고 ‘잘 작성되었다’는 의미로 count를 그대로 리턴한다.
30~35: 버퍼의 끝부터 count만큼 user_buffer에서 복사를 해오는데, 오류가 생길 경우 –EFAULT를 출력한다. 정상 처리된다면 bufferOffset를 count만큼 증가시킨 후 count만큼 잘 읽었다고 값을 반환한다.

```
38  static ssize_t my_read(struct file *filp, char *buffer, size_t buffer_length, loff_t * offset)
39  {
40      size_t length = 0;
41      printk(KERN_INFO "offset : %lld,  buffer_length : %u, bufferOffset : %u\n", *offset, buffer_length, bufferOffset);
42
43      if( *offset >= bufferOffset )
44          return 0;
45      else if( *offset + buffer_length > bufferOffset )
46      {
47          length = min(buffer_length, bufferOffset - *offset);
48          printk(KERN_INFO "buffer Length : %u",buffer_length);
49          printk(KERN_INFO "buferOffset - sOffset = %d",bufferOffset - *offset);
50      }
51      else
52          length = buffer_length;
53      if(copy_to_user(buffer, stringBuffer + *offset, length) ) {
54          return -EFAULT;
55      }
56      printk(KERN_INFO "length : %d",length);
57      *offset += length;
58      return length;
59  }
60
61  static const struct file_operations myproc_fops = {
62      .owner = THIS_MODULE,
63      .open = my_open,
64      .write = my_write,
65      .read = my_read,
66  };
67
68
69  static int __init simple_init(void){
70      printk(KERN_INFO"Simple Module Init!!\n");
71
72      bufferOffset = 0;
73      proc_dir = proc_mkdir(PROC_DIRNAME, NULL);
74      proc_file = proc_create(PROC_FILENAME, 0600, proc_dir, &myproc_fops);
75      return 0;
76  }
77
78  static void __exit simple_exit(void){
79      printk(KERN_INFO"Simple Module Exit!!\n");
80  }
```

38~ : procfs를 읽을 때 호출되는 함수. 호출된 파일 구조체 filp, 출력 결과를 적어주길 원하는 버퍼 buffer, 해당 버퍼에 출력되고자 하는 최대길이 buffer_length, 현재 어디까지 기록이 되었는지 저장이 되어있는, 일종의 파일 포인터 역할을 하는 offset이 매개변수로 들어온다. 반환 값은 write와 동일하게 정상적으로 읽혀진 길이를 반환해줘야 한다.
43 : 버퍼의 최대길이보다 더 먼 index부터 읽고자 할 때 0을 리턴한다.
45~50 : 버퍼의 끝 부분에서 읽을 수 있는 길이가 buffer_length보다 짧을 때, 정확하게 읽은 수 있는 길이만 계산한 후 length 변수에 넣어준다.
52 : 그 외 정상적으로 buffer_length만큼 읽을 수 있는 경우에는 length에 buffer_length를 대입한다.
53~54: copy_to_user를 통하여 매개변수로 들어온 사용자 버퍼에, 내부 버퍼 시작점으로 부터 offset만큼 더한 위치부터 length만큼을 복사해준다. 복사에 실패하면 -EFAULT를 반환해준다.
57 : 최종적으로 offset을 증가시켜 준다. 평소에 일반 프로그래밍을 할 때 읽기 혹은 파일 쓰기 작업 후 파일 포인터의 값이 증가되어 있는 것을 볼 수 있는데 그 작업을 직업 해준다고 보면 된다.

61~66 : f_ops – file operations들을 위에 정의한 함수들로 지정해놓는다.
69~76 : 모듈이 초기화 될 때 /proc/myproc/myproc 이란 이름으로 procfs file이 생성되게 하고, 위에 정의한 f_ops들로 함수가 실행되도록 설정한다.
78~80 : 메모리를 동적할당하거나 커널 리소스를 가져다 사용한 것이 없기 때문에 별도의 리소스 반환 작업 없이 그대로 종료한다.

완성된 커널 모듈을 컴파일 하기 위해 Makefile 파일을 만든다.

```
1   KERNEL_DIR=/home/echo/download/linux-3.0
2
3   obj-m = simplemodule.o
4
5   KDIR :=$(KERNEL_DIR)
6   PWD :=$(shell pwd)
7
8   export CROSS_COMPILE=arm-linux-gnueabi-
9   export ARCH=arm
10
11
12  all:
13      $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules
14
15  clean:
16      @rm -rf *.ko
17      @rm -rf *.mod.*
18      @rm -rf *.cmd
19      @rm -rf *.o
20      @rm -rf *.ko.*
21      @rm -rf modules.order
22      @rm -rf Module.sysvers
23      @rm -rf .tmp_versions
24
```

1 : 디바이스에 사용된 커널이 들어있는 디렉토리로 KERNEL_DIR를 셋팅한다.
8~9 : 크로스 컴파일을 위한 셋팅값을 설정한다.
13 : 커널로부터 커널모듈을 컴파일한다.
15~ : make clean에 해당되는 명령. 작업 디렉토리 내 컴파일로 생겨나는 파일들을 모두 삭제한다.

위와 같은 Makefile을 만든 후, 같은 디렉토리 내에서 make를 실행하면 simplemodule.ko라는 모듈이 생긴다. 그 모듈을 sdb를 이용하여 디바이스에 push한 후, insmod 명령어를 통해 모듈을 추가할 수 있다.

Insmod 후 lsmod로 모듈이 제대로 추가되었는지 볼 수 있고, module init에서 호출한 printk가 정상적으로 호출되는지는 dmesg에서, 그리고 최종적으로 제대로 procfs에 값이 입력 및 출력이 되는지는 아래 명령어로 확인이 가능하다

echo testtext > /proc/myproc/myproc
cat /proc/myproc/myproc

C. nilfs 마운트

nilfs를 마운트하기 위해서는 우선 menu config를 통해 커널이 nilfs를 지원하도록 셋팅을 해야 한다.

이미지 퓨징을 위해 커널을 컴파일 하기 전에 make ARCH=arm menuconfig를 통해 nilfs를 활성화 시킬 수 있다. 터미널에서 위 명령어를 입력한 후 GUI 메뉴가 뜰 경우 filesystem에서 nilfs 항목을 on 시킨 후 메뉴를 종료하면 정상적으로 적용이 된다.

타이젠 내부에서는 nilfs2로 파일시스템의 포맷이 불가능하다. 그래서 호스트PC에서 미리 포맷

후 집어넣어 주는 방식으로 진행하여야 한다.

- 우분투에서 nilfs 를 다룰 수 있게 도와주는 패키지인 nilfs-tools 를 설치한다

sudo apt-get install nilfs-tools

- dd 명령어로 파일 공간을 할당한다. 단, 파일 공간을 비워둬야 하므로 /dev/zero 내부의 값들을 복사하여 할당하도록 하기 위해 아래와 같은 명령어를 사용한다

dd if=/dev/zero of=./diskfile bs=1024 count=200000

(/dev/zero를 입력요소로, ./diskfile위치에 블럭 사이즈 1024byte로 200,000개의 블럭의 파일을 생성)

- mkfs.nilfs2 로 위에서 생성된 ./diskfile 을 nilfs2 로 포맷한다.

mkfs.nilfs2 ./diskfile

- 포맷까지 완료된 diskfile 을 디바이스에 푸시한다

./sdb root on

./sdb -d push ./diskfile /home/developer

(큰 파일 전송 시 디바이스와의 연결이 자주 끊어지는데, 새 터미널에서 ./sdb dlog로 디바이스와 연결중인지 아닌지를 파악하는 것이 좋다. 전송에 대략 10분 이상이 걸리기 때문이다.)

- 전송된 diskfile 을 I/O 디바이스로 등록한다.

losetup /dev/loop0 ./diskfile

- 적당한 디렉토리를 생성 후, 그 디렉토리에 마운트한다.

mount -t nilfs2 /dev/loop0 nilfs_disk/

D. iozone 벤치마크

iozone은 파일 시스템 벤치마크 시스템이다. 블럭 크기, 총 레코딩 크기, 순차적, 랜덤형 Read, Write 등 수많은 옵션들을 제공한다.

우선 iozone을 arm에서 사용하기 위해서 크로스 컴파일을 해야한다. Iozone tar버전을 설치하면 Makefile로 컴파일 할 수 있도록 소스가 제공된다.

make 명령어를 실행하면 지원하는 수많은 플랫폼을 제시하면서 이 중에서 하나로 정하여 make를 실행하라고 나온다. 그 중 우리는 arm이 필요하므로, 아래와 같은 명령어로 크로스컴파일을 한다.

make linux-arm

그 후 생성된 iozone을 sdb push 를 이용하여 /home/developer로 이동시킨다.

Ext3에 대한 벤치마크의 경우는 /home/developer에서, nilfs2에 대한 벤치마크는 3번 항복에서 만든 nilfs_disk 디렉토리 내부로 iozone을 이동시킨 후 작동시키면 된다.

이 과제에서는 다양한 크기의 블럭사이즈, 그리고 충분히 큰 총 쓰기크기로 벤치마크를 하려고 했으나, 타이젠이 불안정하여 포맷한 nilfs 2 파일을 올리는 데에도 운이 안좋을 경우 하루 넘게 걸리는 경우가 많아서, 다양한 실험은 하지 못했다.

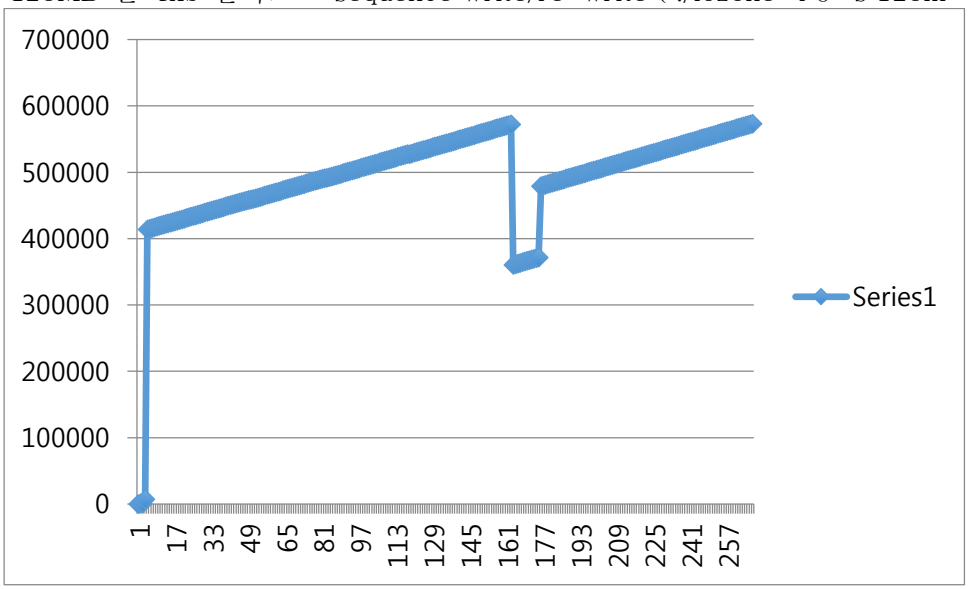
그리고 가장 큰 문제로는 nilfs2로 포맷되어 마운트된 nilfs_disk/ 디렉토리에서 iozone을 동작시키면 iozone이 실행하면서 작성한 파일 조각들이 지워지지 않고 그대로 파일시스템에 남는다. 그래서 nilfs_disk/ 디렉토리 내부에는 아무런 파일도 없지만 df 명령어로 디스크의 남아있는 파일 크기를 보면 전부 다 사용되어서 더이상 벤치마크가 불가능한 경우가 잦았다. 그리하여 본 보고서에서는 ext3 에 대해서는 최대한 다양한 실험을 했지만 nilfs_2에 대해서는 기초적인 벤치마크밖에 진행하지 못하였다.

3. 결과 및 분석

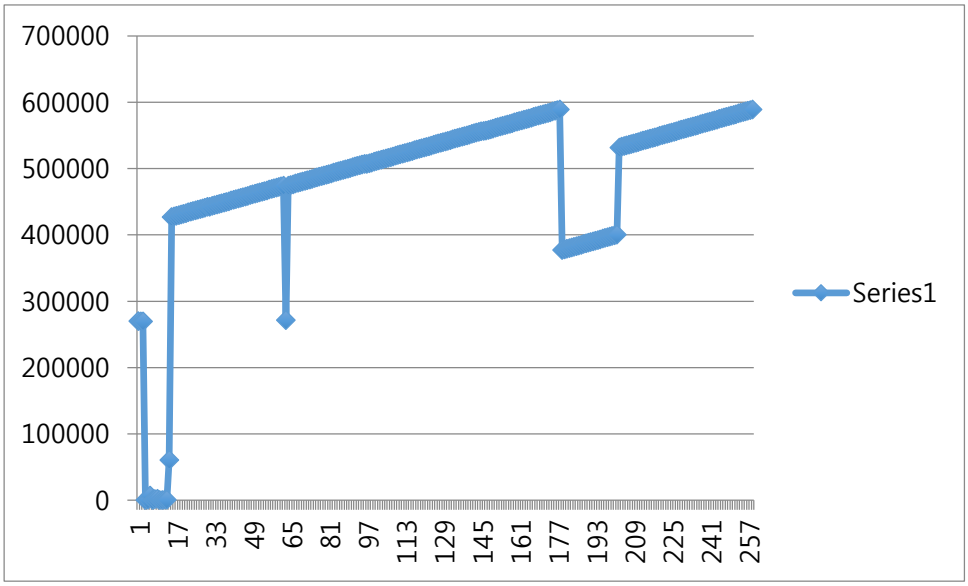
결과 분석에 앞서, block I/O에 대한 time을 로깅하지 않고 그저 block 접근에 대한 index만 로깅했다. 그 이유는 block에 대해 접근한 실제 시간은 본 과제에 크게 중요하지 않고, 오히려 time get 함수 호출로 시스템에 부하가 올 수 있기 때문에 그렇다. block이 전체 순서에서 몇번째로 I/O가 되었는지만 알아도 충분히 Ext3와 nilfs2의 차이점을 알 수 있었다.

A. Ext3

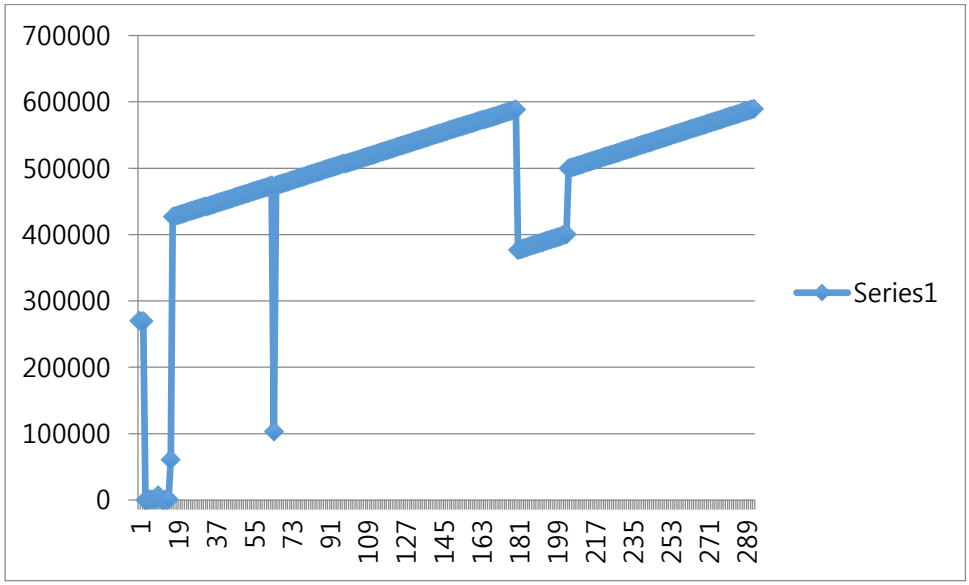
i. 총 128MB 를 4kb 블럭으로 sequence write/re-write (./iozone -i 0 -s 128m -r 4k)



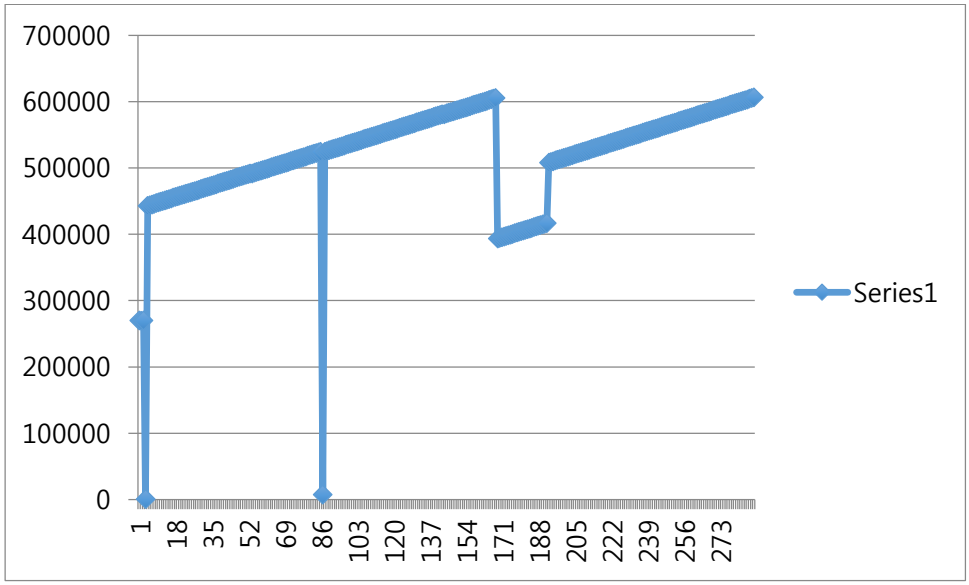
ii. 총 128MB 를 16kb 블럭으로 sequence write/re-write (./iozone -i 0 -s 128m -r 16k)



iii. 총 128MB 를 128kb 블럭으로 sequence write/re-write (./iozone -i 0 -s 128m -r 128k)

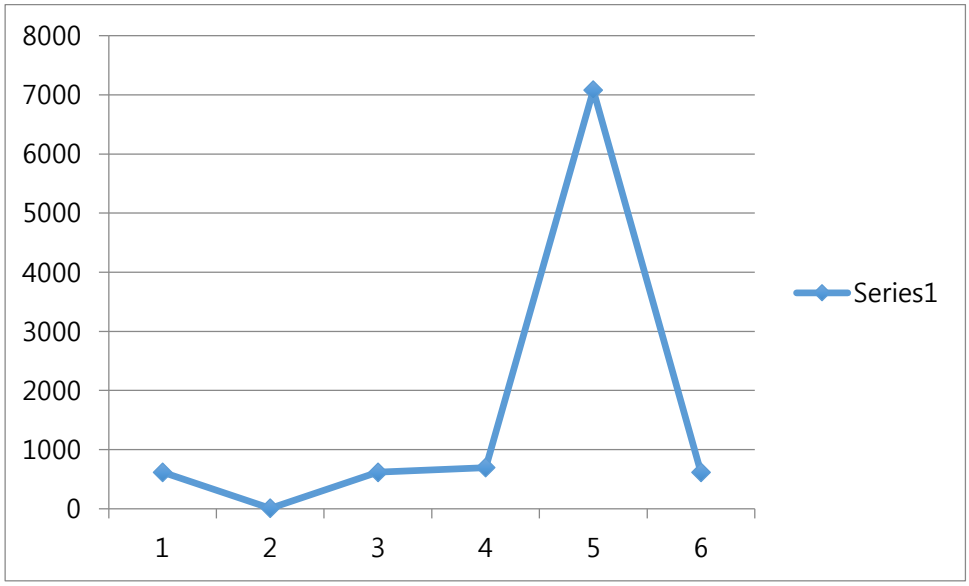


iv. 총 128MB 를 1M 블럭으로 sequence write/re-write (./iozone -i 0 -s 128m -r 1M)

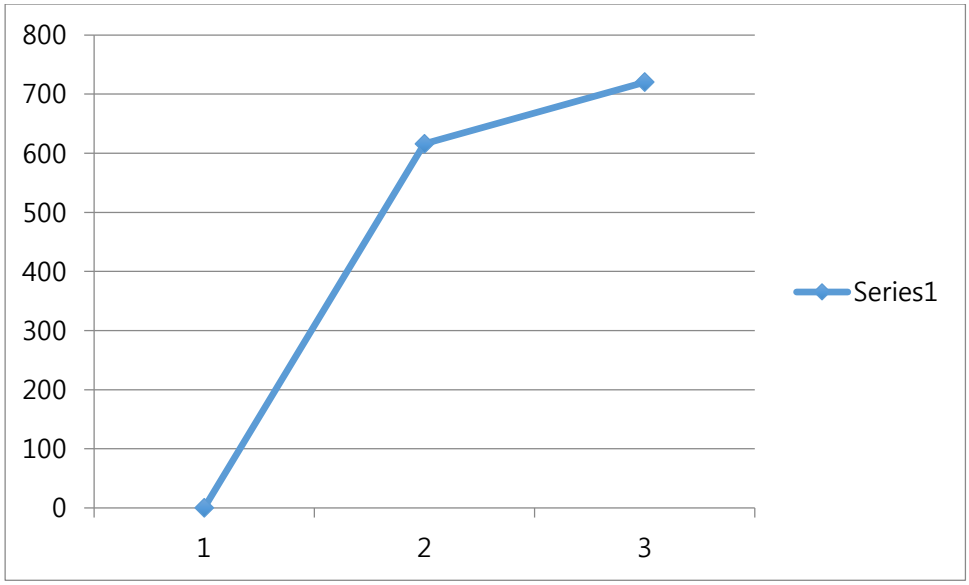


B. Nilfs2

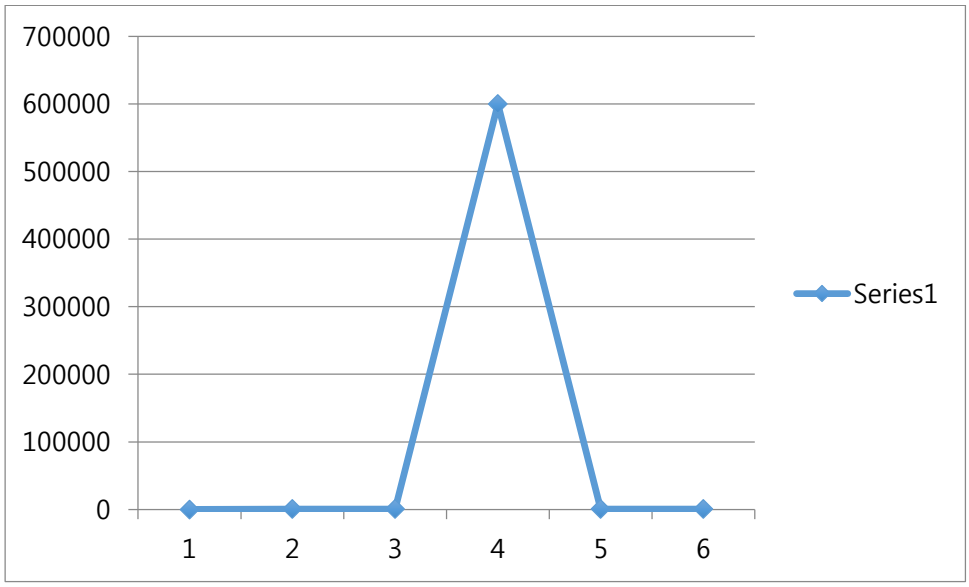
i. 총 32MB 를 4kb 블럭으로 sequence write/re-write (./iozone -i 0 -s 128m -r 4k)



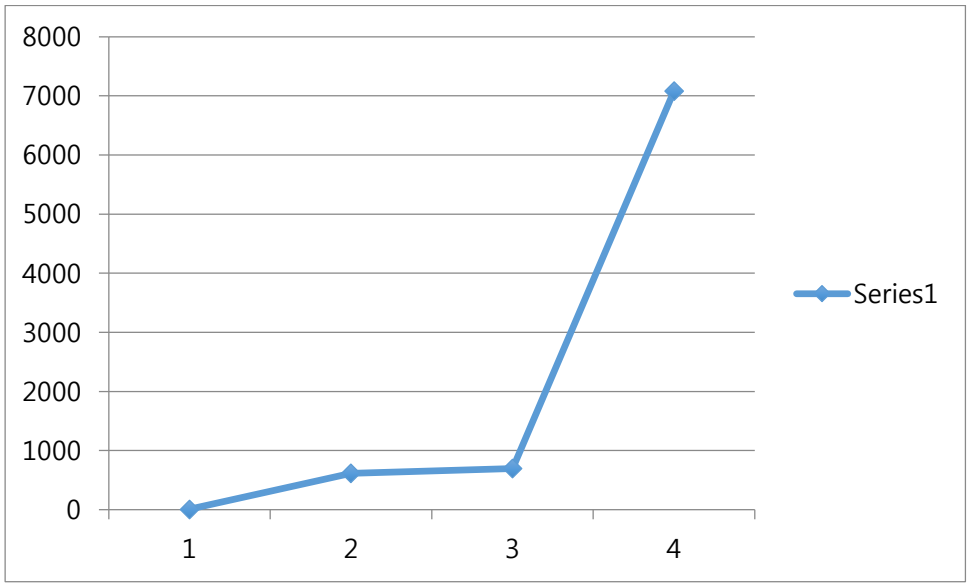
ii. 총 32MB 를 16kb 블럭으로 sequence write/re-write (./iozone -i 0 -s 16m -r 16k)



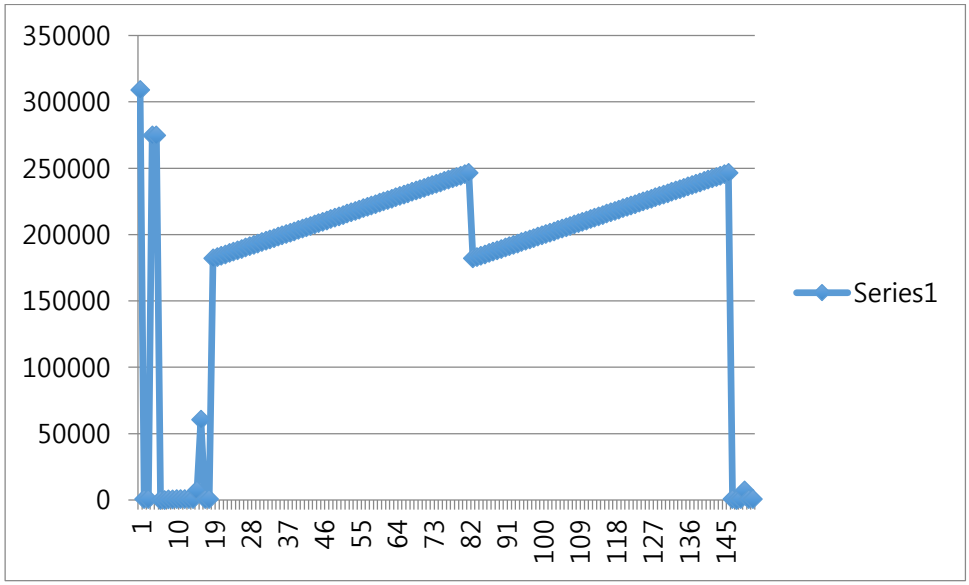
iii. 총 32MB 를 32kb 블럭으로 sequence write/re-write (./iozone -i 0 -s 32m -r 32k)



iv. 총 128MB 를 4kb 블럭으로 sequence write/re-write (./iozone -i 0 -s 128m -r 4k)



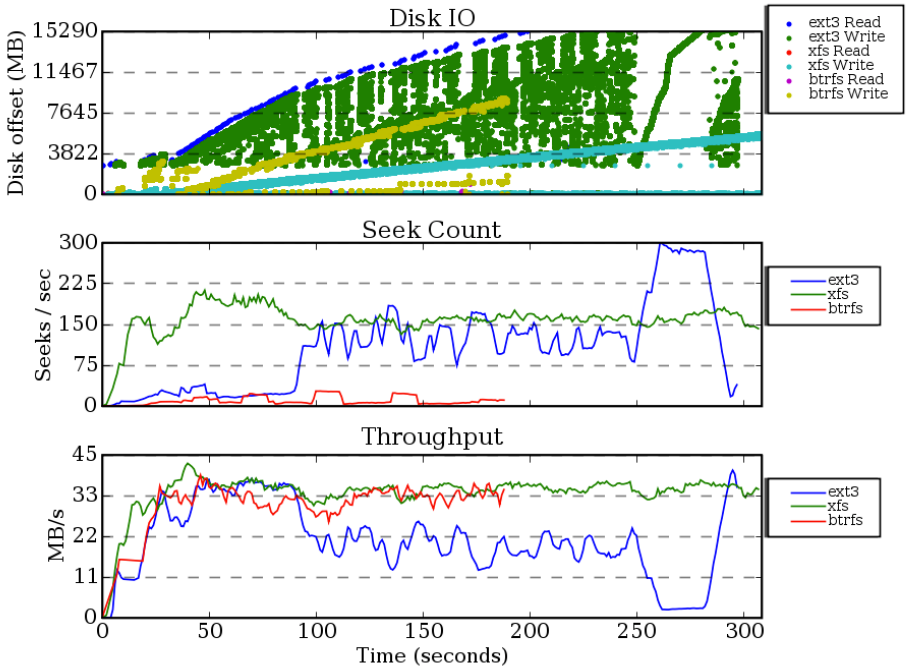
v. 총 128MB 를 4kb 블럭으로 RANDOM write (./iozone –i 4 –s 128m –r 4k)



C. 분석 – Ext3

Ext3의 경우 총 4번의 결과가 거의 동일하게 나왔다. 블럭의 크기를 임의로 지정했지만 내부 bio 구조체에서는 크게 영향을 미치지 않은 것으로 보인다. 수업 시간에 실험 결과 예시로 본 것과는 다르게 크게 Random Access를 하고 있는 것 처럼 보이지는 않는다.

그리하여 혹시 실험이 잘못 설계되었는지 의심이 들어서 관련된 자료를 찾아보았는데, 오라클사의 자료인 아래 사진자료에 보면 초록색 점이 ext3의 write이다. 앞부분은 다르지만 뒷 부분에 하나의 직선으로 나온 것 처럼 결과가 나온 것을 확인할 수 있다. 그 부분은 Throughtput이 굉장히 낮을 때의 결과인데, 본 실험이 128MB를 총량으로 설정했고, 128MB를 테스트하는 데에도 총 20초 이상 걸렸기 때문에, 아래 throughtput 그래프의 뒷 부분처럼 본 과제에서 실행한 iozone의 벤치마크가 충분히 낮은 throughtput이었음을 알 수 있다. 타이젠이 아닌 일반 VM에서 본 실험을 한다면 충분히 일반적인 그래프 모양대로 나올 것을 예상할 수 있다.



(출처 : https://www.google.co.kr/url?sa=i&rct=j&q=&esrc=s&source=images&cd=&cad=rja&uact=8&ved=0CAUQjhxqFQoTCJTdluSU9sgCFYV0pgodnBIHrQ&url=https%3A%2F%2Foss.oracle.com%2F~mason%2Fseekwatcher%2F&psig=AFQjCNFJtjCu0QXwj3tTFFLpDfAPCk-rCw&ust=1446705654249155)

D. 분석 – Nilfs2

Ext3와는 다르게 Nilfs2는 당황스러운 결과가 나왔다. Submit된 bio 객체의 수 자체가 적었다. 처음에는 iozone이 제대로 동작하지 않았나 했지만 마운트된 nilfs_disk/ 디스크의 사용된 용량이 늘어남으로 제대로 동작함을 확인할 수 있었다(iozone을 실행하면 벤치마킹 한 만큼 디스크에 tmp 파일이 작성되는데, iozone이 종료되어도 그 사용공간이 free되지 않는 버그가 있었기 때문).

그에 따라 bio에 관하여 찾아보았더니, 이유를 알 수 있었다. Bio는 연속적인 block I/O에 대해서는 한꺼번에 모아서 처리를 한다고 한다. 수업시간에서 배운대로 Nilfs는 처음부터 sequntial하게 write를 하는데, 그러다보니 블럭의 파편화가 없이 처음부터 작성되어서, submit 되는 bio 수 자체가 줄어든 것이다.

강제로 파편화를 일으키기 위해서 iozone에 random write를 실행시켰다(nilfs2 실험 e). 그랬더니 예상한 대로 ext3와 비슷한 그래프가 그려졌다.

좀 더 정확한 벤치마크 결과가 필요하지만, 크게 분석을 결론지어보면 Nilfs2는 순차적인 I/O에서는 빠른 속도를 지원해줄 수 있을것이라 예상할 수 있다. 일반적으로 커널 함수 콜이 적을수록 성능은 빠르기 때문이다.

4. 과제 시 어려웠던 부분

이 항목에 대해서는 할 말이 매우 많다. VM에서 우분투로 했으면 금방 할 수 있는 과제인데, Tizen과의 VM 사이의 연결이 불안정하여서, 100MB가 넘는 DiskFile을 올리는데 걸리는 시간만 하루 종일이 걸린다. 하루 종일 걸리는 이유는 타이젠이 자기 멋대로 연결을 끊어버리는 경우가 많아서, 게다가 로그로 끊겼는지 아닌지를 제대로 알 수 있는 방법이 없어서 대략 30분 기다려보고 안된 거 같으면 다시 보내보고, 그것을 반복해야 했기 때문이다.

그것 외에도 타이젠에 관련된 자료도 심각하게 없고, 에러가 발생해도 그것에 대한 해결책 솔루션도 많이 나와있지 않아서 굉장히 난감했다. 게다가 한번은 타이젠의 롬 자체가 나가버리면서 하루 종일 아무것도 하지 못하고 과제 Free day를 하루 더 사용해야만 했다.

과제 자체가 타이젠에서만 할 수 있고, 타이젠에서만 배울 수 있는 독특한 과제라면 타이젠을 선택하는 것은 참 좋은 결정이라고 생각한다. 하지만 그것도 아니고, Nilfs도 mount해서 사용할 것이면 우분투에서도 충분히 가능한 숙제인데 불편한 플랫폼에서 과제를 진행한 것에 대한 불만이 매우 많다.

개인적으로는 2차 과제에서는 타이젠 말고 일반적인 리눅스 머신에서 작업을 하도록 해줬으면 하는 마음이다.

5. 비고 사항

2015년 11월 2일 팀원 남세현이 식중독 증상으로 몸이 매우 아파서 하루 동안 과제를 진행하지 못하여서 Free day를 하루 늘려줄 것을 조교님에게 문의했습니다. 처방전을 본 보고서 오프라인 제출물에 동봉해서 보냅니다.