시스템프로그래밍

1차 과제 보고서

목차

1. 개발 환경

하드웨어 : 1.6 GHz 듀얼 코어 Intel Core i5, RAM 4GB/8GB, 동적할당하드디스크 30GB/256GB
가상머신 : Oracle Virtualbox 6.0.18
운영체제 : Ubuntu 16.04 LTS (64bit)
커널 : Linux-4.4.0
파일시스템 : Ext4, F2FS
사용 언어 : C
벤치마크 : iozone3_414

2. 배경지식

파일

Linear array of bytes
파일을 사용하는 순서는 다음과 같다.

1. Open(2)
   int open(const char* pathname, int flags, mode_t mode)
   pathname의 파일을 flags로 연다. 파일이 정상적으로 열렸다면 file descriptor의 양의 정수반환하고, 그렇지 않다면 -1을 반환한다.

2-1. Read(2)
   int read (unsigned int fd, char __user * buf, size_t count)
   fd로 대상 파일에 접근하여 파일을 읽고, 읽은 데이터를 buf에 count만큼 담는다. 읽기가 끝난 위치로 파일 포인터를 옮긴다. 반환 값은 읽어온 byte 수이다.

2-2. Write(2)
   int write (unsigned int fd, char __user * buf, size_t count)
   fd로 대상 파일에 접근하여 파일에 buf에 담긴 데이터를 count만큼 쓴다. 쓰기가 끝난 위치로 파일 포인터를 옮긴다. 반환 값은 쓰여진 byte 수이다.

3. Close(2)
   int close(int fd)
   대상 파일을 닫는다. 닫기가 정상적으로 수행되면 0을 반환하고, 그렇지 않다면 -1을 반환한다.

파일시스템

파일시스템은 파일과 물리디스크 블록 간의 mapping을 제공하는 소프트웨어를 의미하지만 디스크에 들어 있는 파일 전체를 의미하기도 한다. 사용자 영역이 아닌 커널 영역에서 동작하고 파일을 빠르게 읽고 쓰는 기본적인 기능을 원할하게 수행하는 것을 목적으로 한다.

파일 시스템은 일반적으로 여러 개의 계층으로 나누어져 구성된다. 계층 구조를

사용하면 파일 시스템의 코드 중복을 최소화할 수 있다. 계층별 단계는 아래와 같다.

1. Application programs : 명령을 입력한다.
2. Logical file system : 파일 시스템의 메타 데이터를 관리한다.
3. File-organization module: 파일의 논리 블록 주소를 물리 블록 주소로 변환한다. 각 파일의 논리 블록은 0부터 N까지 번호가 주어지고 데이터를 실제 저장하는 물리 블록은 저장장치의 주소이다.
4. Basic file system : 장치 드라이버에게 저장장치의 물리 블록을 읽고 쓰도록 명령을 내린다.
5. I/O control: 장치 드라이버가 저장장치 하드웨어 맞게 명령어를 전달한다.
6. Devices :전달된 명령어를 수행한다.

파일시스템은 디스크 파티션 별로 하나씩 둘 수 있다. 하드 디스크는 필요에 따라 Partition으로 분할 되고 이때 분할된 Partition마다 파일시스템이 하나씩 생성된다. 파일시스템의 역할은 파일의 관리와 보조저장소 관리(저장공간 할당), 파일 무결성 메커니즘, 접근방법 제공이 있다.

파일 시스템은 boot block, super block, inode block, data block으로 분리된 구조를 가진다. 파일 시스템은 이 네가지 영역으로 구성된 자료구조를 제어한다.

1.boot block : 파일 시스템에 커널을 적재 시키기 위해 부팅 시 필요한 코드를 저장하는 영역

2.Super block : 파일시스템에 있는 총 블록의 개수, 블록 크기등 전체 파일시스템에 관한 정보를 가진 영역

3.Inode block : 각 파일이나 디렉토리에 대한 대부분의 정보를 가지고 있는 레코드

4.Data block : 실제 데이터가 저장되는 공간

가상파일시스템(VFS, Virtual File System)

다양한 logical 파일시스템을 abstract함으로써 서로 다른 하드웨어를 사용하는 파일시스템들이 logically 동일하게 동작할 수 있도록 한다. 즉 open(), read(), write() 같은 시스템콜을 호출할 때 각 파일 시스템이나 물리적 매체와 상관없이 동일하게 동작시키는 역할을 한다. 이 때 사용자가 연결된 파일시스템이 어떤 것인지 신경 쓰지 않고 시스템콜을 자유롭게 할 수 있는 이 기능을 제공하는 계층이 파일시스템 추상화 계층이다. 리눅스에서는 파일 시스템을 추상화하기 위해 네가지 객체를 사용한다. 슈퍼블록(SuperBlock), 아이노드(inode), 덴트리(dentry), 파일(file)이 있다.

1. Superblock

슈퍼블록은 각 파일 시스템별로 구현하며 본질적인 파일시스템 메타데이터이다. 여기에는 블록크기, 총 블록 수, 다른 메타데이터 구조체(아이노드 등)의 기본 정보를 포함한다.

슈퍼블록은 매우 중요해서 복사본을 여러 곳에 저장해 놓기도 한다.

2.Inode

아이노드는 파일의 이름을 제외한 해당 파일 각각의 모든 정보를 가지고 있으며 각 파일 이름에 부여되는 고유한 번호이다. 파일형태, 크기,위치 등에 관한 정보가 있고 아이노드 테이블에서 이 번호로 모든 정보를 찾을 수 있다. 또 시스템은 파일이름이 아닌 아이노드 번호로 파일을 처리한다.

3.Dentry

디렉토리 엔트리(Directory Entry)의 약자로 아이노드의 번호와 파일 이름을 관련하여 파일과 아이노드를 연결시켜준다. 또 캐시를 유지하여 자주 접근되는 경로를 더 빠르게 접근할 수 있게 해준다. 마지막으로 디렉토리와 그 디렉토리에 있는 파일들의 관계를 유지하기도 한다. 덴트리 객체는 슈퍼블록이나 아이노드처럼 디스크에 저장되는 것이 아니라 메모리에서 사용자 패턴에 동적으로 생기고 없어진다.

4.File

파일은 프로세스가 사용중인 파일을 표현하는 객체이다. 사용자 공간에서 먼저 보이는 것이 바로 이 파일 객체이다. 파일 객체는 파일이 어디에 저장되어 있는지, 어떤 프로세스들이 사용하고 있는지를 유지한다.

Ext4

Ext4 파일시스템은 Delayed Allocation과 Multiblock Allocation의 두 가지 블록 할당방법을 이용한다. Multiblock Allocation은 Ext4에서 일반적으로 4KB크기의 블록을 여러 개로 묶어서 블록그룹 단위로 관리하는 것을 말한다. 그리고 이 블록그룹은 슈퍼블록, 블록 비트맵, 아이 노드 비트맵, 테이블 블록 등으로 구성된다. 쓰기를 할 때 가까운 블록에 할당하고 쓰기 요청을 순차적으로 처리한다.

Ext4의 장점은 위에서 설명한 Multiblock Allocation을 통해 이전에 쓰이던 블록 매핑 방식 대신 근접한 블록을 하나로 묶음으로써 성능이 향상된다는 점이다. 또 과거의 파일 시스템과 호환성이 유지된다는 점도 장점으로 뽑힌다.

단점은 delayed allocation과 데이터 유실 가능성이다. 지연된 할당은 데이터가 디스크에 기록되기 전에 발생한 시스템 충돌이나 전원 차단에 의해 추가적 데이터 유실 위험을 야기한다.

Log Structured File System(LFS)

로그 구조 파일시스템은 모든 데이터를 순차 쓰기로 기록하는 구조로 플래시 메모리의 성능을 향상시켜 플래시 메모리에 적합하다. 로그 구조 파일 시스템은 각 데이터를 모두 로그에 기록한다. 사용하지 않는 로그는 제거하고, 새로운 로그를 쓸 공간을 위해 Garbage Collection을 한다. 이런 순차쓰기 방식은 플래시 메모리를 포함한 대부분의 장치에서 임의쓰기보다 나은 성능을 보여주므로 LFS가 많이 이용된다.

LFS는 SSD(Solid-State Disk)에 적합한 형식이다. 플래시는 쓰고 지울 수 있는 횟수가 제한되어 있다는 기본적 문제점을 가지는데 로그는 전체 장치의 여러 위치에 균일하게 쓰여진다. 따라서 지우기 주기가 최소화된다. 이 때문에 LFS는 SSD에서 잘 수행되며 우수한 Wear-Leveling 기능을 제공한다.

가장 큰 장점은 속도이다. 앞서 말한 것처럼 데이터를 로그에 순차적으로 기록

하기 때문에 쓰기와 복구에 뛰어나다. 그리고 순차적으로 디스크에 쓰기 때문에 당연히 무작위로 I/O에 쓰는 동작보다 훨씬 빠르다. 찾기에 대한 부담이 제거되고 결과적으로 디스크 속도가 빨라져 전체적으로는 파일 시스템의 속도가 개선된다.

하지만 단점은 Garbage Collection이다. 순차적으로 쓰는 도중 데이터저장 공간이 부족해지면 수행하는 Garbage Collection이 무효 데이터와 유효 데이터가 섞인 공간에서 유효데이터를 다른 공간으로 복사 이동하는데 이 수행 동안 유효데이터가 많을수록 발행되는 비용이 커진다. 그리고 사용자가 느끼는 응답시간도 길어진다.

F2FS

Flash-Friendly File System의 약자로 플래시 기반 파일 시스템이다. 메타데이터는 파일시스템의 앞부분에 저장된다.

데이터를 관리하는 임의 쓰기 영역과 실제 데이터가 저장되는 순차 쓰기 영역으로 나누어 사용한다. 순차쓰기를 할 연속된 여유공간인 Free Segment를 확보하기 위하여 무효화된 데이터를 모아서 Free Segment를 만드는 GC를 파일시스템에서 수행해주어야 한다. Free Segment가 부족하게 되어 GC를 하면, 유효한 데이터를 복사하는 GC 오버헤드때문에 쓰기가 지연되고, 이로 인하여 쓰기성능이 크게 저하될 수 있다.

F2FS는 Checkpointing으로 저장장치와 파일 시스템 간에 일관성을 유지한다. 그래서 체크포인트 영역을 참조하면 가장 최근에 체크 포인팅된 상태로 되돌아갈 수 있다. 시스템이 갑작스럽게 종료되어 체크 포인팅 전에 파일시스템이 종료되면, 마지막으로 체크 포인팅 된 상태로 돌아가서 파일시스템과 저장 장치 간에 일관성을 유지한다.

Proc File System

운영체제의 각종 정보를 커널 모드가 아닌 유저모드에서 쉽게 접근할 수 있도록 만들어 줌으로 시스템 정보에 쉽게 접근할 수 있도록 도와주는 시스템이다. Proc 파일 시스템은 어떤 장치에도 마운트되지 않고 커널 메모리에서만 돌아간다.

실제 proc 파일 시스템을 이용하지 않고 커널 데이터 구조체에서 원하는 시스템 정보를 직접 가져올 수 있다. 하지만 별도의 프로그래밍 과정을 거쳐야 하고 여러가지 요건을 만족시켜주어야 가능하다. 대부분의 경우에는 proc 파일시스템으로만 정보를 가져올 수 있다.

Proc 파일 시스템을 사용하면 파일 시스템 오버헤드를 줄일 수 있다. 각 파일의 inode와 superblock 객체를 관리할 때마다 운영체제에 요청을 하는데, 상당한 오버헤드가 발생한다. Proc 파일 시스템은 이를 해결하기 위해 직접 파일시스템을 관리하므로 오버헤드가 준다.

Bio 구조체와 submit_bio 함수

Bio 구조체는 I/O를 수행할 디스크 영역 정보와 데이터를 저장하기 위한 메모리 영역의 정보를 가지는데 저장영역에 포함된 초기 섹터번호, 섹터의 수, 입출력 연산에 연관된 메모리 영역을 기술하는 한 개 이상의 세그먼트를 포함한다. Submit_bio를 통해 bio를 전달하고 따라서 generic_make_request()가 호출된다.

Loadable Kernel Module(LKM)

Micro-kernel의 장점을 취하며 동적으로 kernel에 load와 unload를 할 수 있다. Monolithic Componets와 loadble Components를 분리한 구조이다. 사용자 공간 프로그램과 같이 생각하면 안 된다. LKM은 커널의 일부부일 뿐이고 그러므로 자유롭게 시스템을 실행/정지할 수 있다.

개발과정은 다음과 같다. 모듈 프로그램 작성 -> 컴파일 -> 로드 -> 동작 확인 -> 제거. 의 과정을 거친다. LKM작성 파일은 일반적으로 Makefile과 소스코드로 이루어진다. kernel의 헤더 파일을 사용하여 작성한다. 반드시 해당버전의 커널에서만 동작이 가능하므로 작성을 위해선 대상 커널의 소스가 필요하다.

LKM의 장점은 커널을 자주 재구축할 필요가 없고 필요할 때에만 메모리에 적재함으로써 메모리를 절약할 수 있다는 점이다.

3. 소스코드 설명

(커널소스위치)/linux-4.4/block/blk-core.c

line39-line43

```
//    writer : Yun Yurim
//    begin modifying
#include <linux/time.h>
#define q_MAX 1000
//    end modifying
```

쓰기가 발생한 시간을 측정하기위해 헤더파일을 추가하고, 원형큐의 크기를 매크로로 선언한다.

line2103-line2130

```
//    writer : Yun Yurim
//    begin modifying

// struct for hw1
//          entry of a circular queue
typedef struct _sphw
{
        const char* fs_name;                    // file system name : ext4 / f2fs
        long time;                              // write time
        unsigned long long block_no;        // block number
}sphw;

sphw c_q[q_MAX];                                // the circular queue
EXPORT_SYMBOL(c_q);                             // for proc file
```

```
int q_front = 0;                                // front of the circular queue
EXPORT_SYMBOL(q_front);                         // for proc file

// function for the circular queue
//           insert sphw at the front of the circular queue
void push_cq(sphw new_sphw)
{
        c_q[q_front] = new_sphw;
        q_front = (q_front+1)%q_MAX;        // circular
        return;
}
EXPORT_SYMBOL(push_cq);                         // for proc file
//    end modifying
```

과제에서 요구하는 정보들을 관리하기 위해 구조체 sphw를 정의하여 멤버로 파일시스템이름(fs_name), 쓰기가 발생한 시각(time), 쓰기 블록 번호(block_no)를 정의한다. sphw를 원소로 갖는 원형큐를 선언하고 해당 자료구조를 EXPORT_SYMBOL처리를 한다. 원형큐를 관리하기 위해 원형큐의 헤드를 가리키기위한 변수 q_front와 원형큐에 원소를 추가하기위한 함수 push_cq를 정의하고, 이들에 대해 EXPORT_SYMBOL처리를 한다. LKM에서는 EXPORT_SYMBOL로 선언된 변수나 함수에 접근하여 필요한 정보를 얻을 수 있다.

line2133-line2202

```
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);

                //    writer : Yun Yurim
                //    begin modifying
                if (rw & WRITE) {
```

```
                count_vm_events(PGPGOUT, count);

                sphw new_sphw;
                if(bio == NULL)
                {
                        printk(KERN_WARNING "No Bio!!\n");
                } else {

                        // get write time
                        struct timespec now_t;
                        getnstimeofday(&now_t);

                        // get file system name
                        //              warning : super block could be NULL
                        if(bio->bi_bdev->bd_super != NULL)
                        {
                                new_sphw.fs_name = bio->bi_bdev->bd_super->s_type->name;
                        } else {
                                new_sphw.fs_name = "";
                                printk(KERN_WARNING "No File System Name!!\n");
                        }

                        new_sphw.time = now_t.tv_sec;

                        // get block number
                        new_sphw.block_no = bio->bi_iter.bi_sector;

                        // push information into circular queue
                        push_cq(new_sphw);

                }
        // end modifying

        } 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);
}
```

쓰기가 발생하여 분기가 발생하는 경우에 대해 파일시스템이름, 쓰기 발생 시각, 쓰기 블록 번호 정보가 담긴 sphw구조체를 원형큐에 추가한다.

bio 포인터가 NULL을 가리킬 때, 해당 포인터가 가리키는 bio 속의 정보에 접근을 시도하면 커널이 죽는 경우가 발생한다. 따라서 bio 포인터가 NULL을 가리키는 경우, sphw구조체에 대한 작업을 수행하지 않는다. bio 포인터가 NULL이 아닌 경우, 쓰기 정보가 담긴 sphw구조체 new_sphw를 원형큐에 추가한다.

쓰기가 발생한 시각을 저장하기위해 timespec구조체 now_t를 선언한다. linux/time.h에 선언된 getnstimeofday함수를 통해 now_t에 현재 시각을 기록한다.

쓰기가 발생한 파일이 속한 파일시스템의 이름을 new_sphw.fs_name에 할당한다. 파일시스템의 이름은 bio가 가리키는 구조체의 멤버인 bi_bdev가 가리키는 block_device 구조체, block_device의 멤버인 bd_super가 가리키는 super_block 구조체, super_block의 멤버인 s_type이 가리키는 file_system_type구조체, 그리고 file_system_type구조체의 멤버인 name이다. 이 과정에서 수퍼블락이 존재하지 않는 경우, 즉 bd_super가 NULL인 경우가 발생할 수 있다. 따라서 분기를 통해 이를 처리한다.

now_t에서 현재 시각을 초단위로 읽어와 new_sphw.time에 쓰기가 발생한 시각을 할당한다.

쓰기 블록 번호를 new_sphw.block_no에 할당한다. 쓰기 블록 번호는 bio가 가리키는 구조체의 멤버 bi_iter가 가리키는 bvec_iter 구조체, bvec_iter구조체의 멤버인 bi_sector이다.

쓰기 정보를 담은 new_sphw를 push_cq함수를 통해 원형큐에 추가한다.

hw/lkm/myproc.c

line1-line17

```
/*
writer : Yun Yurim
*/

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

```c
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/proc_fs.h>
#include <asm/uaccess.h>

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

#define q_MAX 1000

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

선언부이다. proc file 프로그래밍에 필요한 헤더파일들을 추가한다. proc file 파일이 저장될 디렉토리와 proc file 이름을 매크로로 선언한다. 원형큐의 최대 크기를 매크로로 선언한다. proc file을 관리할 포인터 proc_dir과 proc_file을 선언한다.

line19-line33

```c
// to use kernel's circular queue
typedef struct _sphw
{
	const char* fs_name;			// file system name : ext4 / f2fs
	long time;				// write time
	unsigned long long block_no;	// block number
}sphw;


extern sphw c_q[q_MAX];			// the circular queue in kernel
extern int q_front;			// front of the circular queue, also in kernel
extern void push_cq(sphw value);	// function for the circular queue
					//	insert sphw at the front of the circular queue
					//	also in kernel
```

extern을 통해 커널의 심볼들을 참조하기 전, 원형큐의 요소인 sphw 구조체를 커널에 정의된 sphw 구조체와 동일하게 선언한다. extern을 통해 커널의 심볼들을 참조한다. 참조할 심볼들은 원형큐(c_q)와 관련 변수(q_front), 함수(push_cq)이다.

line34-line35

```c
// buffer for copying information proc file in kernel to userspace
char result[q_MAX][100];
```

커널에 저장된 정보를 사용자에게 전달할 때 사용하는 버퍼의 abstraction이다.

line38-line43

```c
// customized open : open proc file
```

```
static int my_open(struct inode *inode, struct file *file)
{
        printk(KERN_INFO "Simple Module Open!!\n");

        return 0;
}
```

사용자화한 open이다.

Line45-line59

```
// customized write : write sphw info to proc file
static ssize_t my_write(struct file *file, const char __user *user_buffer, size_t count, loff_t *ppos)
{
        int i;
        printk(KERN_INFO "Simple Module Write!!\n");

        // Buffering
        //            Pop queue from front to front-1, as there's no information of q_rear
        for(i = q_front ; i != (q_front-1 >= 0 ? q_front-1 : q_front-1+q_MAX) ; i=(i+1)%q_MAX)
        {
                sprintf(result[i], "time : %ld || FS_name : %s || block_no : %llu\n",c_q[i].time, c_q[i].fs_name,
c_q[i].block_no);
        }

        return count;
}
```

사용자화한 write이다. 원형큐에 담긴 정보를 버퍼의 abstraction인 result에 저장(load)한다.

Line62-line74

```
// customized read : read sphw info to proc file
static ssize_t my_read(struct file *file, char __user *user_buffer, size_t count, loff_t *ppos)
{
        printk(KERN_INFO "Simple Module Read!!\n");

        // Read buffered information
        //            Copying information in proc file to userspace
        if(copy_to_user(user_buffer, result, sizeof(result)))
        {
                return -EFAULT;
        }
```

```
        return count;
}
```

사용자화한 read이다. 버퍼로 사용한 result에 담긴 데이터를 전부 user_buffer에 복사(flush)하고 count를 반환한다. 만약 복사되지 않은 바이트가 존재한다면 user_buffer 밖의 공간을 접근하려 했기 때문에 오류가 발생했다는 의미를 더해 -EFAULT를 반환한다.

line76-line82

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

기존의 시스템콜을 사용자화한 시스템콜로 오버로딩한다.

Line84-line93

```
// initialize : make proc file
static int __init simple_init(void)
{
        printk(KERN_INFO "Simple Module Init!!\n");

        proc_dir = proc_mkdir(PROC_DIRNAME, NULL);
        proc_file = proc_create(PROC_FILENAME, 0600, proc_dir, &myproc_fops);

        return 0;
}
```

모듈이 커널에 load될 때 실행된다. Proc file이 저장될 proc directory를 생성하고, 해당 디렉토리에 proc file을 생성한다.

Line95-line104

```
// When dispatching this module, remove all this module made
static void __exit simple_exit(void)
{
        printk(KERN_INFO "Simple Module Exit!!\n");

        remove_proc_entry(PROC_FILENAME, proc_dir);
        remove_proc_entry(PROC_DIRNAME, NULL);

        return;
}
```

모듈이 커널에서 unload될 때 실행된다. 모듈을 커널이 load할 때 생성한 proc file과 directory를 삭제하고, 모듈을 unload한다.

Line106-line112

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

MODULE_AUTHOR("YUNYURIM");
MODULE_DESCRIPTION("It's Simple!!");
MODULE_LICENSE("GPL");
MODULE_VERSION("NEW");
```

모듈이 커널에 load/unload 될 때 실행할 함수를 지정한다.

hw/lkm/Makefile

line1-line10

```
# writer : Yun Yurim

obj-m += myproc.o

KDIR = /usr/src/linux-4.4

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

KDIR에 현재 커널 소스의 경로를 할당한다. LKM은 지정된 커널 소스를 참고하여 컴파일 된다. 컴파일 후 생성될 파일 은 myproc.o, myproc.order, myproc.ko, myproc.mod.o, myproc.mod.symvers 이다. make clean이 실행될 경우 컴파일시 생성된 파일들은 삭제된다.

실행 결과(일부)

Ext4

```
time : 1603965487 || FS_name : ext4 || block_no : 36943872
time : 1603965487 || FS_name : ext4 || block_no : 36945920
time : 1603965487 || FS_name : ext4 || block_no : 36947968
time : 1603965487 || FS_name : ext4 || block_no : 36950016
time : 1603965487 || FS_name : ext4 || block_no : 36952064
time : 1603965487 || FS_name : ext4 || block_no : 36954112
time : 1603965487 || FS_name : ext4 || block_no : 36956160
time : 1603965487 || FS_name : ext4 || block_no : 36958208
time : 1603965487 || FS_name : ext4 || block_no : 36960256
time : 1603965487 || FS_name : ext4 || block_no : 36962304
time : 1603965487 || FS_name : ext4 || block_no : 36962568
time : 1603965488 || FS_name : ext4 || block_no : 36964616
time : 1603965488 || FS_name : ext4 || block_no : 36966664
time : 1603965488 || FS_name : ext4 || block_no : 36968712
time : 1603965488 || FS_name : ext4 || block_no : 36970760
time : 1603965488 || FS_name : ext4 || block_no : 36972808
time : 1603965488 || FS_name : ext4 || block_no : 36974856
time : 1603965488 || FS_name : ext4 || block_no : 36976904
time : 1603965488 || FS_name : ext4 || block_no : 36978688
time : 1603965488 || FS_name : ext4 || block_no : 36980736
time : 1603965488 || FS_name : ext4 || block_no : 36982784
time : 1603965488 || FS_name : ext4 || block_no : 36984832
time : 1603965488 || FS_name : ext4 || block_no : 36986880
time : 1603965488 || FS_name : ext4 || block_no : 36988928
```

F2FS

```
time : 1603981826 || FS_name : f2fs || block_no : 1172448
time : 1603981826 || FS_name : f2fs || block_no : 1172696
time : 1603981826 || FS_name : f2fs || block_no : 1172944
time : 1603981826 || FS_name : f2fs || block_no : 1173192
time : 1603981826 || FS_name : f2fs || block_no : 1173440
time : 1603981826 || FS_name : f2fs || block_no : 1173688
time : 1603981826 || FS_name : f2fs || block_no : 1173936
time : 1603981826 || FS_name : f2fs || block_no : 1174184
time : 1603981826 || FS_name : f2fs || block_no : 1174432
time : 1603981826 || FS_name : f2fs || block_no : 1174680
time : 1603981826 || FS_name : f2fs || block_no : 1174928
time : 1603981826 || FS_name : f2fs || block_no : 1175176
time : 1603981826 || FS_name : f2fs || block_no : 1175424
time : 1603981826 || FS_name : f2fs || block_no : 1175672
time : 1603981826 || FS_name : f2fs || block_no : 1175920
time : 1603981826 || FS_name : f2fs || block_no : 1176168
time : 1603981826 || FS_name : f2fs || block_no : 1176416
time : 1603981826 || FS_name : f2fs || block_no : 1176664
time : 1603981826 || FS_name : f2fs || block_no : 1176912
time : 1603981826 || FS_name : f2fs || block_no : 1177160
time : 1603981826 || FS_name : f2fs || block_no : 1177408
time : 1603981826 || FS_name : f2fs || block_no : 1177656
time : 1603981826 || FS_name : f2fs || block_no : 1177904
time : 1603981826 || FS_name : f2fs || block_no : 1178152
```

4. 실행 방법

1. Mount ext4

a. 다음 명령어를 실행하여 가상 디스크를 만든다.

```
dd if=/dev/zero of=~/ext4_disk.img bs=500MB count=1
```

b. 다음 명령어를 실행하여 해당 디스크를 Ext4로 format한다.

```
mkfs -V -t ext4 ext4_disk.img
```

c. 다음 명령어를 차례로 실행하여 Ext4 디스크를 디렉토리에 마운트한다.

```
mkdir ~/ext4_dir
sudo mount ext4_disk.img ~/ext4_dir
```

2. Mount F2FS

a. 다음 명령어를 실행하여 가상 디스크를 만든다.

```
dd if=/dev/zero of=~/f2fs_disk bs=1024 count=600000
```

b. 다음 명령어를 실행하여 해당 디스크를 F2FS로 format한다.

```
mkfs.f2fs ~/f2fs_disk
```

c. 다음 명령어를 실행하여 F2FS 디스크를 I/O device로 등록한다.

```
sudo losetup /dev/loop7 ~/f2fs_disk
```

d. 다음 명령어를 차례로 실행하여 F2FS 디스크를 디렉토리에 마운트한다.

```
mkdir ~/f2fs_dir
sudo mount -t f2fs /dev/loop7 ~/f2fs_dir
```

3. 커널 코드 컴파일 및 설치

a. /usr/src/linux-4.4/block/block-blk.c를 ~/hw/block-blk.c로 대치한다.

b. ~/hw/compile_kernel.sh, hw/install_kernel.sh를 /usr/src/linux-4.4/로 옮기고 /usr/src/linux-4.4/에서 다음 명령어를 차례로 실행한다.

```
bash compile_kernel.sh
bash install_kernel.sh
```

4. LKM loading

a. ~/hw/lkm으로 이동 후 다음 명령어를 실행한다.

```
make
```

실행 결과, myproc.ko가 생성된다.

b. 다음 명령어를 통해 root권한을 얻는다.

```
su
```

c. 다음 명령어를 통해 myproc을 커널에 load한다.

```
insmod myproc.ko
```

d. 모듈이 커널에 load된 결과 다음의 디렉토리와 파일이 생성된다.

```
/proc/myproc
/proc/myproc/myproc
```

5. 쓰기 측정

a. 파일 시스템의 특징에 따라 쓰기 동작 시나리오를 구성한다.

b. 다음 명령어를 실행하여 iozone3_414를 이용해 쓰기 테스트를 진행한다.

```
Case : ext4
~/iozone3_414/src/current/iozone -i 0 -f ~/ext4_dir/anyfile
Case : F2FS
~/iozone3_414/src/current/iozone -i 0 -s 50M -f ~/f2fs_dir/anyfile
```

6. write to myproc

다음 명령어를 실행하여 /proc/myproc/myproc에 쓰기를 수행한다.

```
echo ext4 > /proc/myproc/myproc
```

7. read myproc

다음의 명령어를 실행하여 /proc/myproc/myproc을 읽고, 로그를 저장한다.

```
cat /proc/myproc/myproc > ~/log.txt
```

8. 데이터 정재

~/log.txt의 크기가 매우 클 수 있으므로 다음의 명령어를 통해 ~/log.txt에서 필요한 만큼만 데이터를 얻는다.

Case : ext4

```
head -n 1000 ~/log.txt > ~/ext4_result.txt
```

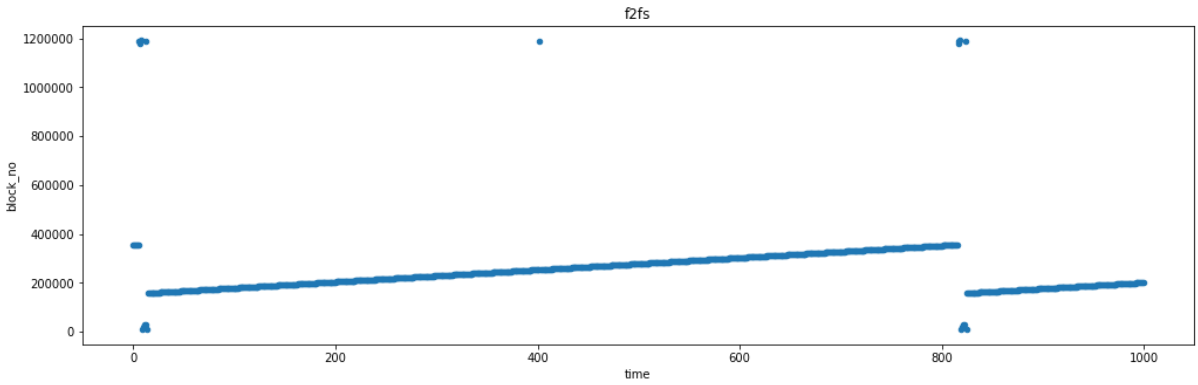
Case : F2FS

```
head -n 5000 ~/log.txt > ~/f2fs_result.txt
```

결과 파일에서 fs_name이 f2fs인 데이터 1000개를 추출한다.

5. 실험결과 및 분석

1. F2FS 결과 그래프

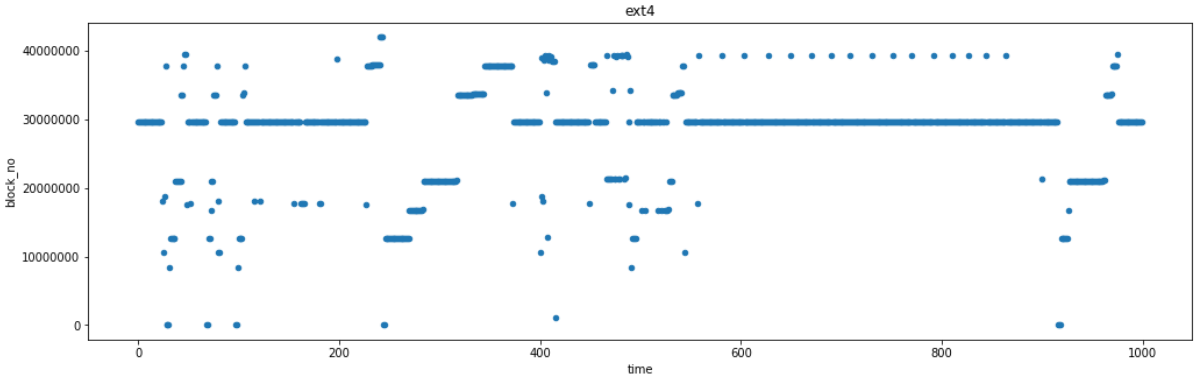


[그림 1]

F2FS, [그림1]의 그래프는 선형적이다. 이는 순차적으로 새로운 블록을 할당하고 있음을 의미한다. 일반적인 로그 구조 파일 시스템은 append-only 쓰기 정책을 사용한다. 그래서 쓰기 작업이 요청되면, 마지막으로 할당했던 블록의 바로 다음 위치에 새로운 블록을 할당하는데 이를 위의 그래프가 잘 보여주고 있다.

1. 쓰기 요청이 들어오면 블록을 할당하고 뒤이어 들어오는 요청들은 순차적으로 이어서 새로운 블록을 할당해준다.
2. 800번 시간대에서 순차적 할당을 하지 않고 다른 블록들로 이동하는 것이 보인다. 이는 해당 공간을 모두 사용했거나 그 외 설정된 조건이 만족되었기 때문이다.
3. 이 때 0번대의 슈퍼 블록 첫번째 페이지로 이동한다. 기존에 있던 LFS들과 달리 메타데이터들을 다른 데이터들과 구분하여 파일시스템 맨 앞부분에 저장하기 때문이다.
4. 저장된 여분영역 inode를 찾아서 유효한 블록 위치로 이동한다. 다음 영역으로 이동시에 슈퍼블록의 여분 영역에 다음 작업영역의 시작 블록 번호를 기록한다. 이것이 [그림1]에서 순차적 할당을 하다가 연결이 잠시 끊어져 보이는 이유이다.
5. 이동한 블록에서 다시 순차적인 블록 할당을 실행한다.
6. 이를 반복하면서 계속 선형적 write를 한다.

2. Ext4 결과 그래프



[그림 2]

Ext4, [그림2]는 인접한 블록에 데이터를 저장한다. block이 30000000 주변에서 계속 쓰기 할당을 하고 있음을 볼 수 있다. 그러다가 free 페이지의 수가 얼마 안 남으면 Stale 페이지에서 garbage collection을 시작하기 때문에 기존의 블록을 이탈하는 것처럼 보인다. 조금 더 자세하게 정리해본다.

1. write 연산이 수행되면 파일 시스템은 이것을 일정 시간동안 캐시에 유지하고 일정한 양이 차면 연속된 공간에 즉시 데이터가 위치할 블록으로 할당한다.
2. 캐시에 일정량을 모아둬서 크기가 크지만 싱글 블록 할당 대신 다중블록 할당을 통해 해결한다.
3. 30000000에서 쓰기를 할당하다가 더 이상 공간이 충분하지 않다고 판단하면 garbage collection을 수행한다. 이 때 ext4의 extents 매핑 방식으로 인해 블록 여기저기에 움직이는 것을 볼 수 있다.
5. gc를 진행하고 돌아와서 다시 블록 그룹단위로 쓰기 요청을 처리하고 이를 계속 반복한다.

3. 결과 비교

두 그래프의 공통점은 둘 다 순차적 블록 할당을 하는 것이다. 두 시스템의 파일 할당 단위는 다르지만 둘 다 순차적으로 진행한다.
두 그래프의 차이점은 더 이상 할당할 블록이 없다고 판단했을 때 나타난다. f2fs 파일 시스템은 슈퍼블록의 아이노드 정보를 통해 새로운 블록을 찾아가고, Ext4 파일 시스템은 garbage collection을 통해 빈 블록을 생성하고 다시 연속된 블록으로 돌아와서 데이터를 저장한다.

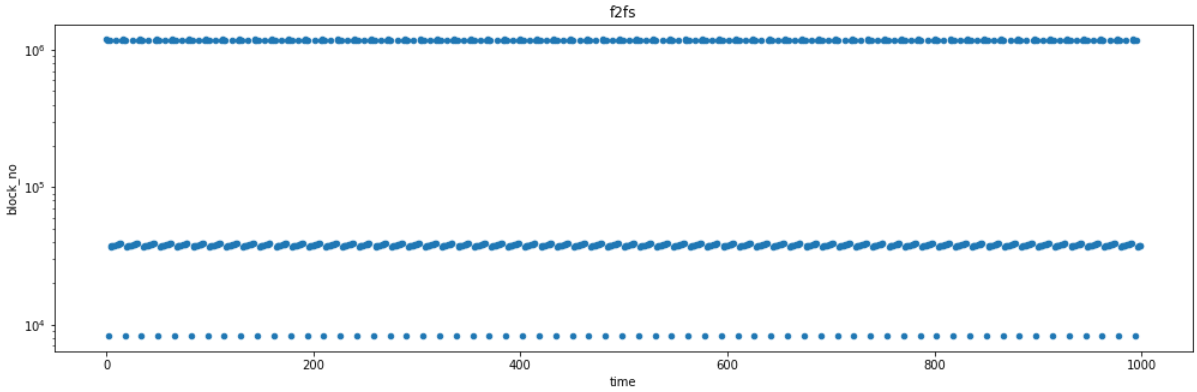
6. 과제 수행 시 어려웠던 부분과 해결방법

□ F2FS 실험

리눅스의 기본 파일시스템은 Ext4이다. F2FS 실험을 진행하기 위해서는 가상디스

크를 생성하고 해당 디스크를 F2FS로 포맷한 후, 해당 디스크를 I/O디바이스로 등록하고 새로운 디렉토리에 마운트해야한다. 이때 이 디렉토리는 전체 파일 시스템에서 유일한 F2FS로 F2FS에 대한 로그와 Ext4에 대한 로그가 proc file에 기록되었다. 따라서 F2FS실험에 대한 그래프를 생성할 때, 추가적인 프로그래밍을 통해 로그 raw 파일에서 F2FS의 로그만을 추출해야했다.

벤치마킹툴로 F2FS 실험을 진행할 때, 블록사이즈를 512KB로 너무 작게 결정하여 다음과 같이 F2FS가 가진 LFS의 특징을 잘 보여줄 수 없는 그래프를 생성했다.



블록사이즈를 50MB로 키우며 실험시나리오를 수정했고 본문에 첨부한 그래프와 같은 결과를 얻을 수 있었다.