Log Structured File System Profiling

comparison with ext4

개발 환경

가상머신 - Oracle VirtualBox 6.1.0

Guest Machine - Ubuntu 16.04.6 LTS

Host Machin - macOS Catalina 10.15.7

사용 언어

C언어

커널 버전

Linux 4.4.0

하드웨어 스펙

CPU : I9-9980H RAM : 32GB (Guest: 2 Core, RAM : 8GM)

1. 배경 지식

1차 과제는 플래시 메모리에서 주로 사용되는 Log-structured file system인 F2FS와 리눅스의 기본 파일 시스템인 EXT4의 쓰기(write) 동작 시 블록 할당 분석 및 비교이다.

1) 플래시 메모리

플래시 메모리는 데이터를 지우고 다시 기록할 수 있는 기억 장치다. 플래시 메모리는 seek time을 줄이고, crash recovery가 빨라서 많은 저장장치로 활용되고 있다. 대부분의 플래시 메모리는 NAND 플래시를 사용하고, 읽기와 쓰기의 단위는 페이지이다. NAND 플래시 메모리에는 여러 개의 블록들이 있고, 각 블록은 32개의 페이지가 있다. 그리고 페이지마다 데이터가 있는 main array 공간과, 오류 검출과 같은 예비 용도를 위한 spare array 공간으로 이루어져있다. 쓰기와 읽기 동작은 페이지 단위인데, 플래시 메모리의 삭제 단위는 페이지가 아닌 블록이다. NAND 플래시 메모리는 쓰기 동작을 하기 전에 삭제부터 해야 하는데, 블록마다 많은 페이지가 있기 때문에 쓰고 있는 페이지와 같은 블록에 속한 다른 페이지의 복사가 필요다. 이 과정을 해결하기 위해서 메모리 플래시에는 Flash Transition Layer이 존재한다.

2) Flash Translation Layer (FTL)

FTL은 파일 시스템 계층에서 플래시 메모리를 디스크처럼 사용하고 디스크와 플래시 메모리의 mismatch를 해결하기 위해 만들어졌다. FTL은 STL, BML, LLD로 3개의 구성으로 나누어져 있다. Sector Translation Layer(STT)는 wear leveling과 페이지의 garbage collection을 관리한다. 쓰기 동작 수행 시, FTL은 wear-leveling을 고려한 새로운 플래시 페이지를 할당한다. Wear-leveling은 곧 지워질 특정 블록들에 수명연장을 고려하는 것이다. 다음으로, 파일의 블록을 플래시 페이지로 매핑을 하며, 과거 페이지를 stale로 바꾸고 새로운 페이지로 매핑 테이블 갱신한다. FTL은 쓰기 작업 시 매핑 테이블의 내용을 변경해서 이전 블록들을 stale로 바꾼 후 신규 블록에 기록해서 플래시 메모리 전체를 균등하게 사용한다. 그리고 free 페이지의 수가 얼마 안 남았을 때 stale 페이지를 garbage collection으로 처리한다. Garbage collection은 블록을 실제로 삭제하지 않고, 삭제 표기만 해두고 필요할때 삭제를 처리하는 것이다. 실제 처리는 BML에서 한다. Bad Block Management Layer(BML)는 garbage collection을 통해 stale 페이지를 많이 가진 블록을 선택하며, 유효 페이지를 다른 블록에 복사하고, stale로 표기된 블록을 삭제하고 재사용하는 기술이다. 마지막으로 Low level Drive(LLD)는 NAND 플래시를 사용하기 위한 드라이버다. FTL은 지우기의 연산 속도를 줄여서, 연속적인 쓰기 동작 패턴에 대해 빠른 처리를 가능하게 해준다.

3) EXT4 & F2FS

EXT4는 ext3의 업그레이드된, FTL을 사용하는 리눅스의 기본 file system이다. 이번 1차 과제에서 사용하는 Ubuntu도 기본적으로 EXT4를 사용한다. 하지만 플래시 메모리의 성능을 최대한 활용하기 위해 삼성은 2012년에 플래시 메모리용 F2FS 파일 시스템을 개발했다. F2FS(Flash-Friendly File System)은 NAND 플래시 메모리의 특성을 고려한 log-structured file system이다. EXT4와 F2FS의 가

장 큰 차이점은 F2FS가 사용하는 log-structured file system이다.

4) Log-Structured File System(LFS)

Log-structure file system은 storage를 연속된 log로 처리하는 file system이다. 데이터와 metadata는 log 처럼 순차적으로 기록된다. LFS의 목적은 파일의 쓰기와, crash 복구의 속도를 향상시키는 것이다. LFS는 디스크의 asynchronous 쓰기를 한 번에 모아서 처리하고, 새로 시작하는 쓰기를 파일에 append 한다. 새로운 쓰기에는 데이터, 인덱스 정보, 디렉터리가 포함되어있다. LFS의 장점은 모든 내용이 한 곳에 같이 있고, log로 연속적으로 저장되기 때문에 seek time을 최소화한다는 것이다. Log 사이에는 빈 공간이 없고, 새로운 데이터가 들어오면 끝에 있는 log에 붙인다. Linux file system 은 metadata의 업데이트는 동기적으로 처리되지만 데이터는 비동기적으로 처리되기 때문에 consistency를 체크하기 위해 전체 디스크를 스캔해야한다. 또한 디스크의 용량 크기에 따라 crash recovery 속도가 너무 많이 걸리는데, LFS는 crash가 발생할 시, 파일의 제일 뒤에 있는 정보(tail)만 확인하면 되기 때문에 효율적이다. LFS에서는 data가 update 될 때 새로운 inode를 오른쪽(tail)에 배치하고 앞에 있는 수정된 데이터는 garbage collection 통해 삭제되어서, crash 발생시 모든 데이터를 스캔 할 필요 없고, tail만 확인한다.

LFS log-structure에는 inode의 위치가 log에 있다. Linux filesystem에 존재하지 않은 inode map를 도입해서 각 inode의 현재 위치를 유지한다. Inode map은 메모리에 caching하고, 읽기 동작 시 inode를 inode map에서 찾는다. LFS는 디스크를 segments로 나누어서 segment cleaning을 통해 free space를 관리한다. Segment마다 Sumamry 블록이라는 헤더가 있고, 헤더에는 다음 Summary블록을 연결하는 포인터가 존재한다. 이 포인터의 segments들은 LFS가 사용하는 log-structure chain으로 연결해준다. Segment cleaning은 라이브 데이터를 다른 segment로 복사하고, segment를 메모리로 읽고, 라이브 데이터만 정리해서 디스크 segment로 사용하는 것이다. 그리고 cleaner를 통해 라이브 데이터 아닌 stale 데이터를 garbage collect 한다. F2FS는 VFS 호출을 제공하기에 충분한 여유 segment가 없을 때 정리를 수행한다.

5) Virtual File System(VFS)

Virtual file system(VFS)은 파일에 대해서 읽기/쓰기의 system call이 호출되면 파일이 속한 파일 시스템의 읽기/쓰기가 호출될 수 있도록 function pointer를 사용한다. VFS의 장점은 시스템의 여러 개의 다른 file system이 사용되더라도, 마치 한개의 파일 시스템만 있는 것처럼 프로그램이 가능하는 것이다. 원래 장치 application에서 읽기가 호출되면 VFS가 읽기/쓰기 전달을 받은 인자를 통해 data의 위치 정보인 device, 블록 번호, size로 바꾸어 buffer cache에 해당하는 block이 있는지 찾는다. Buffer cache에 이미 cache된 블록이 있으면 VFS에게 반환하고, 없을 시 file system에게 요청하여 그 block을 자신의 cache entry에 추가한다. 그리고 VFS는 application에서 인자로 넘겨준 buffer에서 읽어온 블록들을 복사하고 return 한다. 쓰기 동작에는 읽기 동작과 buffer cache에서 찾는 과정까지는 일치하는데, 찾고자 하는 블록이 없으면 file system을 통해 블록을 먼저 디스크에서 가져온다. 쓰기 동작에는 kworker가 쓰기를 수행한 블록에 대해서 dirty bit을 점검하여 변경한 내용이 디스크에 반영되게 한다.

F2FS의 경우 log structure에 모든 내용이 있다. F2FS를 사용할 경우 메모리에는 log structure 하나 밖에 없고 indexing을 통해서 inode를 찾을 수 있다. VFS에서 쓰기 동작을 수행하면, VFS는 inode map을 통해 inode의 정보를 가져오고, 디렉터리 또는 데이터를 찾을 수 있다. 디스크처럼 kworker 통해 디스크와 buffer cache 블록 내용을 synchronize 할 필요가 없으며, log structure는 segment cleaning 통해 필요 없는 데이터를 처리하고 바뀐 내용을 log 끝에 추가한다.

2. 커널 소스

1) blk-core.c

먼저 block과 관련된 정보를 저장하기 위한 circular queue를 구현하였다. circular queue는 충분한 block 정보를 저장하기 위하여 1024개의 block 정보를 저장할 수 있도록 하였고 queue의 front와 rear를 기준으로 새로 저장될 데이터의 index를 지정하였다. queue가 가득 찰 경우 queue의 front를 한 칸 전진시켜 새로운 공간을 마련한다. 이를 통해 계속해서 enqueue가 가능하도록 구현하였다.

blk_core.c 파일의 submit_bio 함수를 수정했다. submit_bio 함수는 파일의 write가 수행될 때 호출되는 함수로 해당 함수에서 사용되는 bio 구조체에 block I/O와 관련된 정보가 존재한다. 해당 구조체의 멤버를 조사하여 bio->bi_iter.bisector가 저장하고 있는 block number와 bio->bi_bdev->bd_super->s_type->name가 저장하고 있는 파일 시스템의 이름을 circular queue에 저장하도록 하였다.

F2FS 파일 시스템에 대한 실험의 경우, EXT4 파일시스템의 로그와 중복 저장되는 것을 막기위하여 파일 시스템의 이름이 F2FS일 경우에만 circular queue에 저장되도록 하였다.

2) block_log_lkm.c - Linux Kernel Module

proc 파일시스템과 Linux Kernel Module을 활용하여 circular queue에 저장된 정보를 저장하고 출력하도록 하였다.

먼저 Linux Kernel Module이 시작되면 proc 파일 시스템에 block log를 저장할 수 있는 디렉토리와 파일을 생성하도록 하였고, echo 명령어를 통해 해당 파일을 write할 경우 미리 선언된 buffer에 circular queue의 정보를 write하도록 하였다. 이후 cat 명령어를 통해 해당 파일을 read할 경우 buffer에 저장된 block log 정보를 user space로 불러오도록 하였고 read된 log를 별도의 log 파일에 저장하였다. read의 경우 반복해서 읽는 것을 막기 위해 한 번 read가 수행된 이후에는 return 값으로 0이 반환되도록 하였다. 이후 Linux Kernel Module을 종료할 경우 remove_proc_entry() 함수를 통해 생성한 디렉토리와 파일을 삭제하도록 하였다.

3) Makefile

Linux Kernel Module을 컴파일하기 위한 Makefile을 작성하였다. 커널 디렉토리 위치를 수정한 커널의 위치로 지정해주었고 clean할 경우 생성된 파일들을 모두 삭제하도록 작성하였다.

3. 실행 방법

EXT4

1) 커널 소스 컴파일

새로 다운로드 받은 linux-4.4 커널 소스의 /block/blk-core.c 파일을 새로 작성한 코드로 수정한 뒤 커널 이미지 컴파일과 커널 모듈 컴파일, 커널 이미지 설치를 수행한다. 이후 해당 커널로 부팅될 수 있도록 grub을 수정한 뒤 재부팅한다.

make bzImage -j [number of CPU]

make modules –j [the number of CPU]

sudo make modules_install –j [the number of CPU]

sudo make install

2) Linux Kernel Module 코드 컴파일 및 실행

Makefile을 이용하여 Linux Kernel Module을 컴파일하고 생성된 모듈을 실행한다.

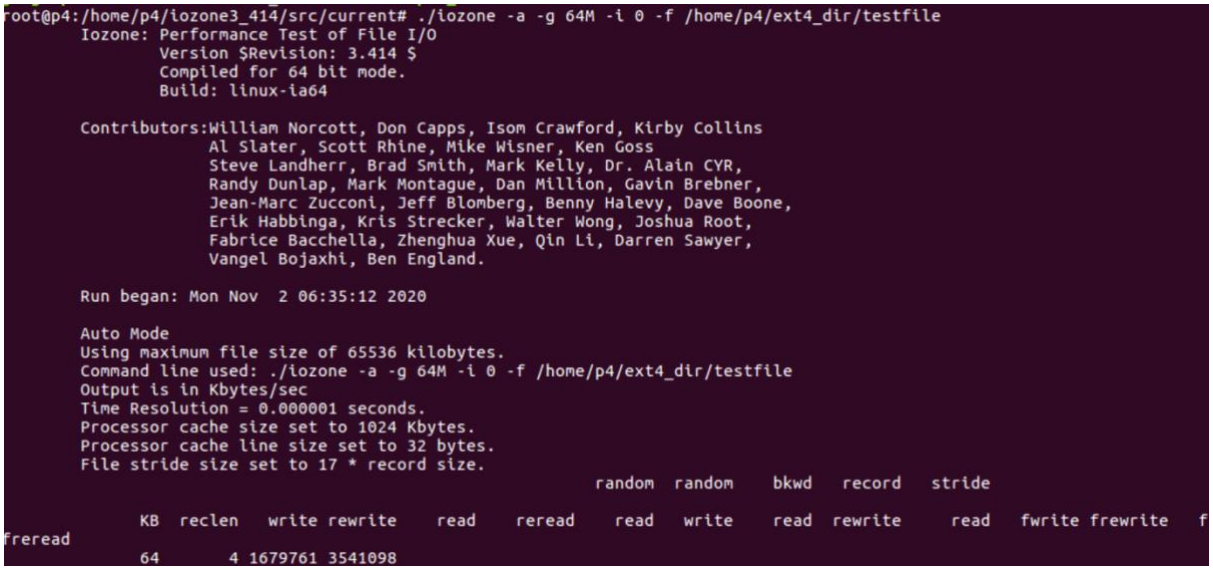
make

insmod block_log_lkm.ko

3) iozone 실행

block log 생성을 위해 iozone을 사용하여 write를 수행한다. -a 옵션을 통해 자동 모드로 수행하고 -g 옵션을 통해 생성될 disk의 size에 맞추어 최대 64M 용량의 파일로 test가 수행되도록 한다. write에 대한 테스트만 진행했기 때문에 -i 0 옵션으로 write/rewrite에 대한 I/O를 수행하도록 했다.

./iozone -a -g 64M -i 0 -f /home/p4/ext4_dir/testfile



4) proc의 write 호출

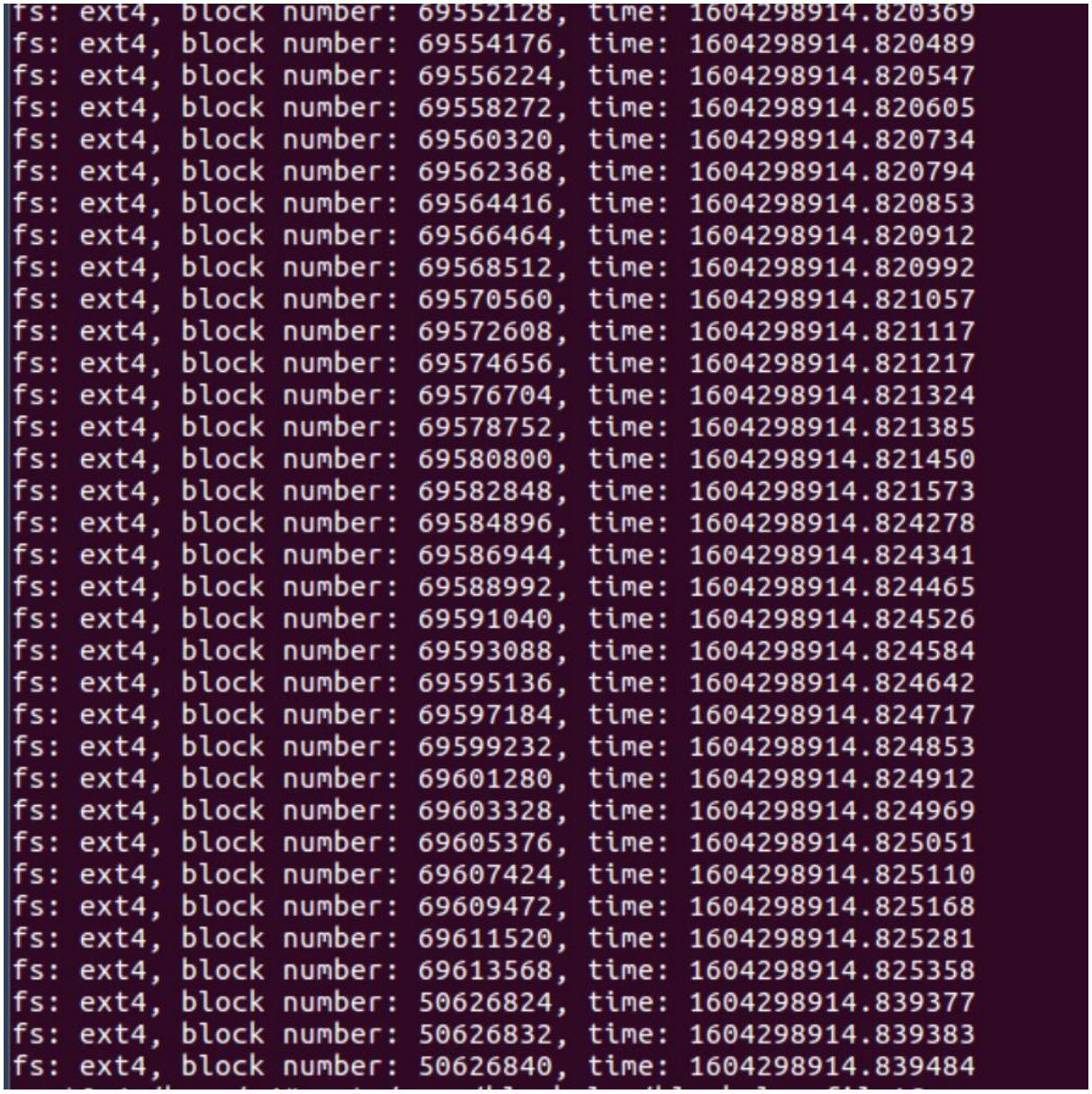
echo 명령어를 통해 proc의 쓰기 연산을 수행해 block log가 proc의 buffer에 저장되도록 한다.

echo "ext4" >> /proc/block_log/block_log_file

5) proc의 read 호출 및 log파일 생성

cat명령어를 통해 proc의 read 연산을 수행해 buffer에 저장된 block log를 출력하고 별도의 log 파일에 저장한다.

cat /proc/block_log/block_log_file >> ext4_log.txt



F2FS

1) 커널 소스 컴파일

커널 소스의 blk_core.c 코드를 F2FS용으로 바꿔주고 컴파일 한 뒤 재부팅한다.

2) F2FS 파일 시스템 마운트

F2FS 파일 시스템을 위해 200MB 크기의 가상 디스크를 생성하고 지정된 디렉토리에 마운트한다.

dd if=/dev/zero of=./diskfile bs=1024 count=200000 - 가상 디스크 생성

mkfs.F2FS ./diskfile - F2FS로 디스크 포맷

losetup /dev/loop7 ./diskfile - 이미지 파일을 블록 device로 등록

mount –t F2FS /dev/loop7 /home/p4/F2FS_dir - 디렉토리에 파일시스템 마운트

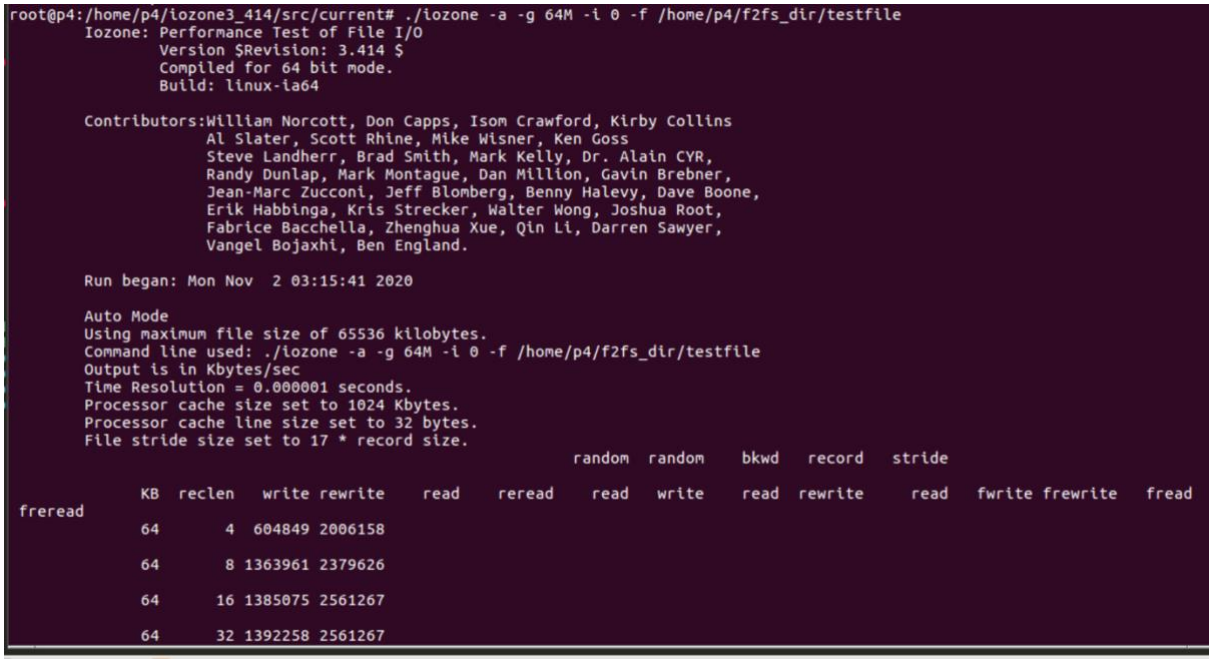
3) Linux Kernel Module 코드 컴파일 및 실행

EXT4와 마찬가지로 Linux Kernel Module을 컴파일하고 module을 실행시킨다.

4) iozone 실행

EXT4와 마찬가지로 iozone을 통해 쓰기 연산을 수행한다.

./iozone -a -g 64M -i 0 -f /home/p4/f2fs_dir/testfile



5) proc의 write 호출

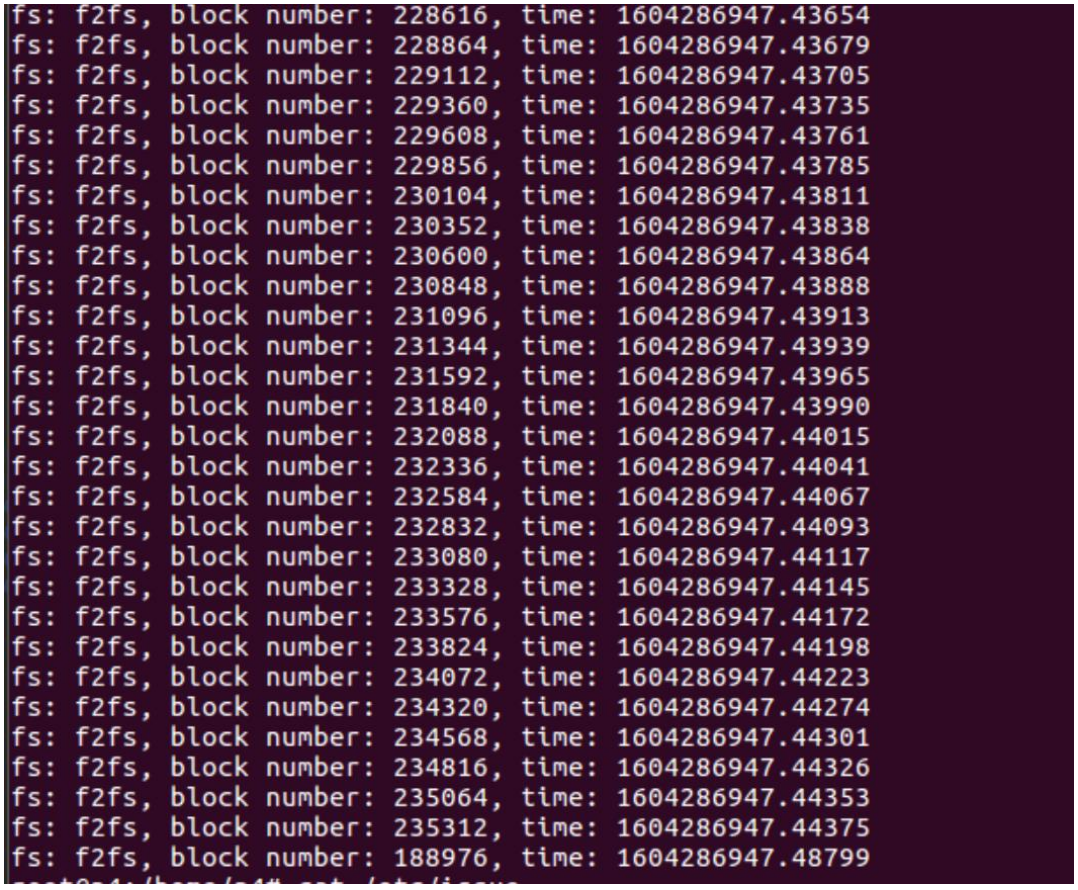
EXT4와 마찬가지로 proc의 write를 호출한다.

echo "f2fs" >> /proc/block_log/block_log_file

6) proc의 read 호출 및 log파일 생성

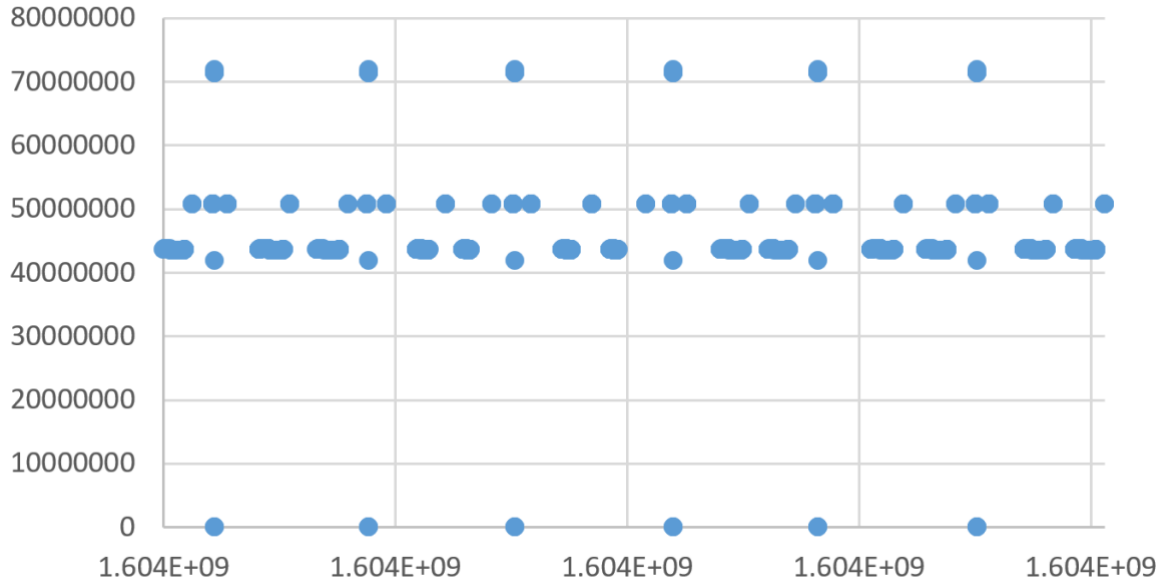
EXT4와 마찬가지로 read를 수행하고 log파일을 생성한다.

cat /proc/block_log/block_log_file >> f2fs_log.txt

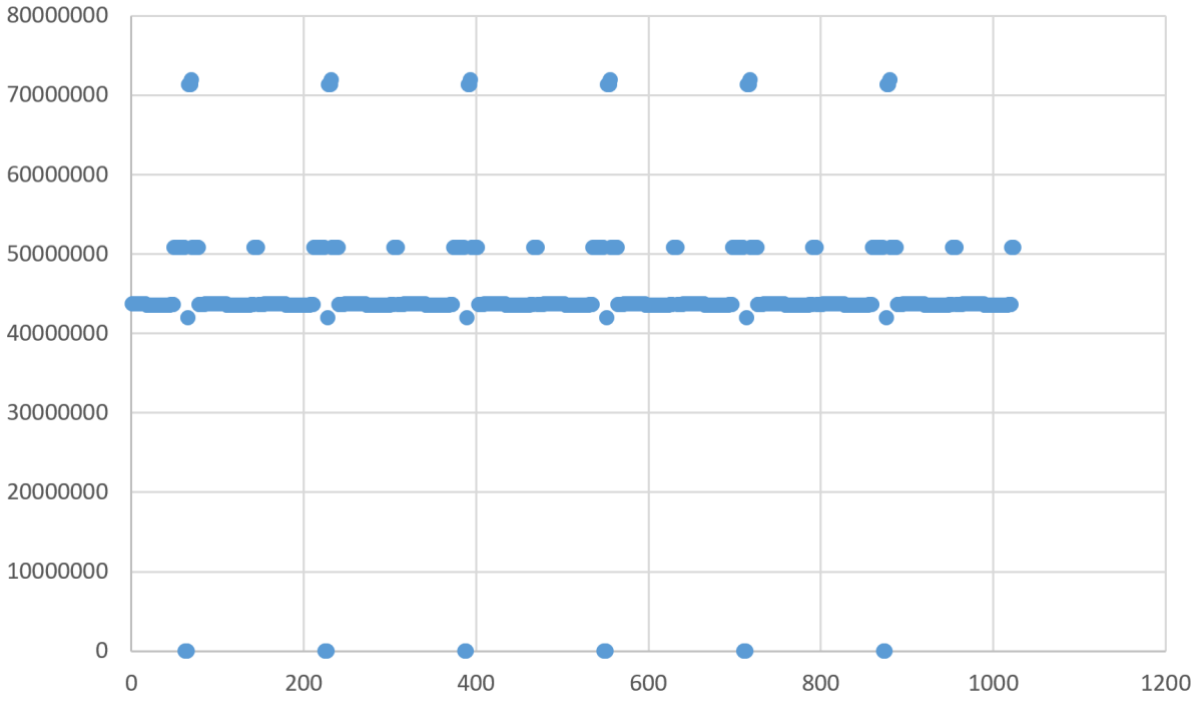


4. 결과 그래프 및 원인 분석

1) EXT4

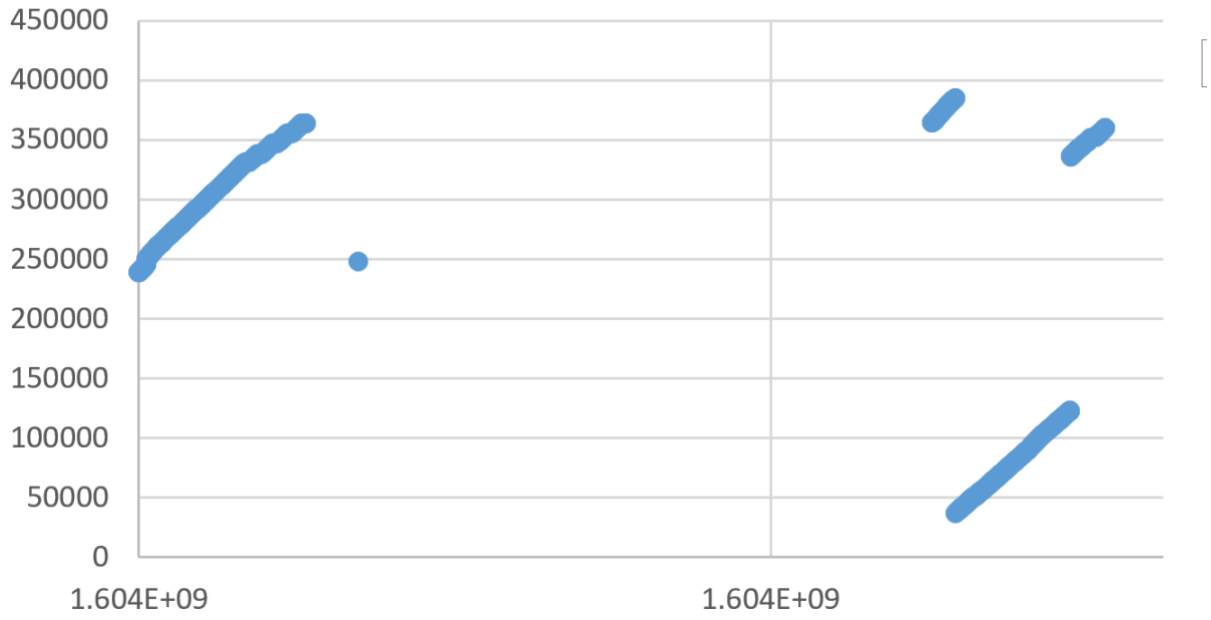


<Figure 1 EXT4 by time>

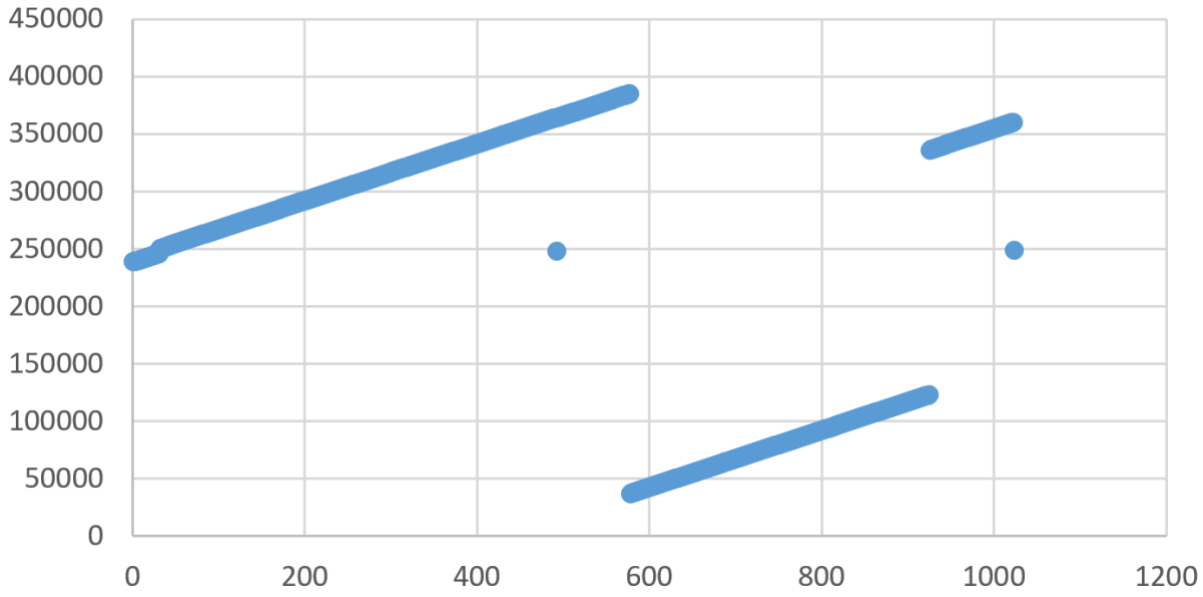


<Figure 2 EXT4 by order>

2) F2FS



<Figure 3 F2FS by time>



<Figure 4 F2FS by order>

F2FS의 경우 쓰기 연산 시 block number가 sequential하게 증가하는 경향이 있음을 알 수 있다. 이는 연속된 block에 새로운 data를 write하는 LFS의 특징을 보여준다. EXT4의 경우 sequential하게 block number가 증가하지 않는다. 또한 EXT4의 경우 같은 block을 여러 번 write하는 것을 알 수 있는데, EXT4의 경우 같은 file에 접근할 경우 해당 file이 저장된 block을 다시 접근하기 때문이다. 반면 F2FS는 rewrite할 경우 새로운 block에 할당한다.

F2FS의 경우 중간에 쓰기를 수행하는 block number가 완전히 바뀌는 부분이 있음을 알 수 있는데, 이것은 다음 block 공간이 free space가 아니기 때문에 이를 건너뛰고 다른 free space에 저장

되는 것임을 유추할 수 있다. LFS는 이러한 free space를 찾는 과정을 Threading과 Copying을 사용하여 수행한다. 또한 LFS는 지속적으로 free space를 확보하기 위해 cleaning을 수행한다. F2FS by time 그래프에서는 중간에 큰 시간 term이 있음을 알 수 있는데, 이러한 시간 term이 있기 전 연속되지 않은 block을 한번 write하는 것을 알 수 있다. 이러한 시간 term의 원인은 새롭게 할당할 block이 존재하지 않을 때 발생하는 F2FS의 garbage collection time과 관련된 것으로 추측하였다.

5. 과제 수행 시 어려웠던 부분과 해결 방법

1) 전체적인 구조 이해

처음 과제를 시작하기 앞서 전체적인 구조와 수행 과정을 이해하는데 많은 시간이 소요되었다. 리눅스 커널 코드를 수정해본것은 처음이었기 때문에 커널 코드에서 생성된 data를 불러오는 과정이 이해가 잘되지 않았다. 강의 자료를 복습하고 몇 번의 시행착오 끝에 리눅스 커널 코드에서 생성된 data가 커널 모듈과 proc 파일시스템을 통해 불러와지고 user space로 복사하여 출력되는 일련의 과정을 이해할 수 있었다.

2) 커널 소스 컴파일 시 null pointer 처리

blk-core.c 코드를 수정한 뒤 커널을 컴파일하고 해당 커널로 부팅을 시도했을 때 부팅이 되지 않았다. 화면에 출력된 에러 코드를 확인해보니 수정한 submit_bio 함수와 관련된 오류가 출력되어 있었기에 수정된 코드에 문제가 있다는 것을 알게되었다. 구글링과 강의 자료를 통해 null pointer에 대한 처리가 필요하다는 것을 알게되었고, circular queue에 block 정보를 저장하기 전에 해당 정보가 존재하는 지를 확인하는 조건문을 추가하였다.

3) Linux Kernel Module 생성 시 proc 파일 삭제

커널 모듈 코드를 수정하고 모듈을 다시 올리려고 할 때 rmmod 이후 insmod를 다시 수행하면 모듈을 init할 때 proc 파일시스템에 생성된 디렉토리와 파일이 삭제되지 않아 정상적으로 모듈이 올라가지 않았다. 이후 구글링을 통해 커널 모듈의 exit 함수에 생성한 디렉토리와 파일을 삭제해야된다는 것을 알게되었다. 그러나 디렉토리와 파일을 정상적으로 삭제하기 위해 parent proc_dir_entry를 각각의 인자로 추가해주는 것을 모르고있었고 삭제 과정에 오류가 생겨 proc 파일 시스템 자체가 멈추는 문제가 발생하였다. 이후 remove_proc_entry의 document를 확인해 정상적인 인자를 추가해주었고 init 과정에서 생성된 디렉토리와 파일이 삭제되도록 하였다.

4) cat을 통해 proc read시 read가 무한히 반복되는 문제

커널 모듈을 통해 생성된 proc 파일시스템에 block log를 write한 이후 cat을 통해 read를 수행했을 때 read가 종료되지 않고 계속해서 반복되는 문제가 발생하였다. 관련된 문제를 검색해보니 read 함수가 return값으로 양수를 반환하여 buffer에 read할 data가 남아있다고 인식될 경우 cat은 계속해

서 read를 수행한다는 것을 알게되었다. block log buffer에 데이터가 저장되어 있을 경우 한번만 read가 수행되도록 첫 번째 read 수행 시에 offset을 block log의 size로 할당하여 두 번째 read 수행에는 offset값을 확인하여 0이 return 되도록 하였다.