고려대학교
KOREA UNIVERSITY

시스템프로그래밍
1차과제 보고서

- Log-structured File System profiling -

제출일	2020.10.31.	**학과**	컴퓨터학과
과목	시스템프로그래밍	**팀**	4
Freeday	2일 사용	**팀원**	홍성윤(2016320187) 윤상준(2016320186)

○ 조원 정보 및 역할

- 2016320187 홍성윤(조장) : 보고서 작성, 큐 및 함수 구현, 구조체 분석, 쓰기 시나리오 구성, 디버깅
- 2016320186 윤상준(조원) : 구조체 분석, 모듈 및 큐 구현, 디버깅, 그래프 작성, 보고서 수정

○ 개발환경

가상머신 : Oracle VM VirtualBox
운영체제 : Ubuntu 16.04 LTS
커널 : Linux-4.4.0
언어 : C

○ 배경지식

1. 파일시스템

- 파일시스템은 버퍼캐시, 디렉토리, 파일과 물리공간의 간 매핑 등 파일에 효율적으로 접근하고 수정하기 위한 추상화된 구현체를 말한다. 파일시스템은 메타데이트를 관리하고 파일과 물리디스크 불록 간의 mapping을 제공함으로써 사용자가 편리하게 파일에 접근하고 사용할 수 있게 해준다.

2. 디스크 기반 파일시스템 구성방식

▷ 개요

ext4 파일시스템은 inode를 통해 파일을 관리하고 파일에 접근한다. inode는 file에 대한 data block index를 계층 형태로 관리하는 방법이다. inode는 file에 대한 속성을 나타내는 field, 물리 블록을 바로 가리키는 direct index과 파일 크기가 커짐으로써 direct index로만은 부족한 추가적인 물리 블록을 저장하기 위한 indirect index table 들로 구성되어 있다. 계층 형태로 index를 관리하기 때문에 파일의 크기가 커져도 inode의 크기는 달라지지 않는다는 장점이 있다.

▷ 구성 및 방식

ext4 파일시스템에서 inode들은 inode table block에 모아져서 저장되고, data block들도 같은 영역에 모아져서 저장된다. super block 구조체는 마운트 된 파일시스템 당 하나씩 주어지고, inode table의 시작위치를 가르킨다. ext4 파일시스템은 디스크 접근 시 seek time을 줄이기 위해서 같은 디렉토리에 속하는 파일과

inode를 같은 cylinder 그룹에 할당함으로써 성능을 높인다.

파일에 write를 하게 되면, 해당 data block 뿐만 아니라 해당 디렉토리와 metadata도 수정해줘야 한다. ext4 파일시스템에서는 data block과 inode의 위치가 나누어져 있기 때문에 write를 수행하기 위해서는 여러 개의 디스크 블록에 접근해야 한다. 디스크의 여러 곳에 접근하며 쓰기를 수행하기 때문에 crash가 발생했을 경우, 디스크의 어느 곳을 쓰고 있었는지 알 수 없다. 따라서 ext4에서 recovery를 위해서는 디스크의 모든 정보를 확인해 consistency를 유지해야한다.

3. lfs 파일시스템

▷ 개요

f2fs 가 기반하고 있는 파일시스템인 LFS 는 모든 디스크 write를 한 번에 모아서 처리하고, 로그형식으로 붙여서 쓰기만 한다. 이는 지우기 속도가 느려 데이터를 수정하지 않고 다른 블록에 계속 쓰는 플래시 메모리의 특성과 잘 맞는다. data, inode, directory 정보를 포함한 write 할 모든 block을 한번에 붙여쓰기 때문에 디스크 기반의 파일시스템처럼 inode와 data block이 나누어져 있지 않다.

▷ 읽기 및 쓰기 동작 방식

lfs 파일시스템에서는 로그형식으로 write를 한다. overwrite 없이 계속 이어서 쓰기만 하므로 seek time을 최소화 할 수 있다. 디스크 기반 파일시스템에서는 파일이 수정되면 inode를 수정하지만, lfs 에서는 수정이 아닌 다시 쓰기를 하므로 같은 파일에 대한 inode가 여러 개 생기게 된다. 따라서 현재 버전의 inode들에 대한 정보를 가지고 있는 새로운 자료구조가 필요하다. inode map 은 가장 최근 버전의 inode들의 위치정보를 가지고 있다. 따라서 lfs에서 write를 수행할 때, 마지막에 inode map도 같이 write해줘야한다.

lfs 파일시스템은 append only 방식으로 쓰기 효율을 높였지만 read 수행이 어렵다. 디스크 기반 파일시스템에서처럼 inode 가 고정된 위치에 있지 않고, 계속해서 새로운 버전이 생기기 때문이다. 따라서 접근하고자 하는 inode의 위치를 찾기 위해서 먼저 inode map에 접근한 뒤 해당 inode에 접근하게 된다. inode map을 캐싱하면 성능향상을 기대할 수 있다.

append only로 인해 디스크에 crash가 발생할 경우 recovery가 비교적 간단하다. 로그 형태로 파일을 저장하므로 파일의 제일 뒤에 있는 정보만 체크하면 된다. 그 전에 있는 로그들은 다 안전하게 동기화가 되었을 거기 때문이기 때문이다.

▷ free space mananament

lfs 에서는 다시 쓰기를 하지 않고 계속 쓰기만을 하기 때문에 시간이 지나면 디스크의 남은 공간이 없어진다. 따라서 free space management를 통해 free space

를 가능한 크게 유지하는 것이 매우 중요한 문제이다. lfs에서는 threading과 copying 두 가지 방식을 사용해 free space를 가능한 큰 단위로 유지하게 한다. segment cleaning 이란 lfs의 쓰기 단위인 segment에서 live한 data를 다른 곳에 모으고 clear한 segment를 만드는 것을 말한다. lfs에서는 append only로 인해 디스크 공간이 금방 부족해 질 수 있으므로 segment cleaning이 매우 중요하다. segment cleaning을 위해서는 제일 먼저 몇 개의 segment들을 메모리로 읽고, live data를 정리해서 모은 후 다른 segment 에 live data를 저장한다.

▷ bio 구조체

bio 구조체는 blk_types.h에 정의되어 있고, bio에는 bvec_list에 대한 정보를 저장하고 있다. bi_io_vec은 bvec_list의 첫 번째 요소를 가르키고 있으며, bi_iter 구조체에는 현재 bvec_list의 index에 대한 정보(index는 bvec_list에서 진행된 위치를 의미합니다.)와 함께 그 해당 bvec의 sector number와 I/O가 얼마나 진행 되었는지 등에 대한 정보가 저장되어 있다. 여기서 bvec은 buffer cache로 요청된 I/O에 대한 최소 단위이며 그에 관련한 정보를 포함하고 있다.

○ 커널 소스를 작성한 부분에 대한 설명

1. myCircle.h

: q_entry 구조체를 정의한 헤더파일. 구조체가 여러 번 정의되는 것을 방지하기 위해 따로 헤더파일을 작성하고 blk-core.c 파일과 작성한 커널 모듈 파일에 include 하여 사용했습니다. q_entry 구조체의 멤버로는 fsys_name, blk_num을 정의해서 해당 write 요청의 파일시스템 이름과 block number 값을 가질 수 있게 했고, timestamp_sec와 timestamp_msec를 통해 요청이 발생했을 때의 timestamp 값을 저장하도록 했습니다.

3. blk-core.c

: circular queue에 데이터를 저장하도록 실제로 수정한 파일. myCircle.h에서 정의한 q_entry 구조체를 사용해 정보를 저장하기 위한 circular queue를 q_entry 구조체 배열로 정의했습니다. 두 파일시스템의 쓰기 요청 처리 방식을 비교하기 위해 각각에 사용될 circular queue를 하나씩 정의하고 각각의 queue의 현재 index를 나타내기 위한 인덱스 변수도 정의했습니다. 정의된 네 개의 변수를 EXPORT_SYMBOL 함수를 이용해서 커널모듈에서 사용할 수 있게 했습니다.

queue에 필요한 정보를 넣기 위해 push함수를 정의했습니다. bio 구조체를 분석한 결과, bio의 멤버인 bi_bdev 구조체에 super block에 관한 정보가 있다는 것을 확인했습니다. linux/fs.h에 정의된 super_block 구조체에 struct file_system_type s_type 이라는 멤버가 있었고 해당 구조체는 파일시스템의 이름 및 파일시스템의

여러 정보를 가지고 있었습니다. file_system_type 구조체의 const char * name을 통해 파일시스템의 이름을 queue에 저장했습니다.

block number 정보는 bio의 struct bvec_iter bi_iter 멤버에서 확인했습니다. bvec_iter 구조체에 bi_sector를 통해 섹터 주소를 가져와 8을 곱해서 4KB 단위의 block number 바꿔 queue의 blk_num에 저장했습니다

포인터의 null 값 참조로 인한 커널 패닉을 방지하기 위해 bio 포인터와, bio 의 bi_bdev 포인터, bi_bdev의 bd_super 포인터가 모두 null 이 아닐 때만 해당 구조체에 접근해 데이터를 queue 에 push 하도록 했습니다. 해당 요청이 어떤 파일시스템의 요청인지에 따라 해당하는 queue에 데이터를 저장하도록 했습니다. 두 개의 queue 모두 queue 가 가득 차면 index를 0으로 초기화해 버퍼의 시작 지점부터 다시 저장하도록 했습니다.

두 개의 queue 배열(mycircle1, 2)와 각각의 현재 인덱스(ptr_num1,2)는 전역변수로써 정의하였으므로 push함수에서 접근이 가능합니다. 따라서 push 함수는 bio구조체 포인터만 매개변수로 받게 했습니다. submit_bio 함수에서 read/write 요청이 있으면 if(rw&WRITE){..} 부분이 실행되므로 해당 IF문에서 push 함수를 호출해 queue에 데이터를 저장하도록 했습니다.

3. 모듈 c파일

: 두 개의 circular queue를 정의해서 두 개의 파일시스템에 대한 정보를 저장하기로 했기 때문에 각각의 queue에 대해서 내용을 출력하는 두 개의 모듈을 작성했습니다. 두 모듈은 이름과 다루는 queue 만 다르고 기본적인 흐름은 같습니다.

먼저 proc 파일 시스템에 파일을 생성하기 위해서 proc_dir_entry 변수를 정의합니다. hello_proc_init 함수에서 해당 모듈이 적재되면 proc 파일과 디렉토리가 생성되도록 하였고 hello_proc_exit 함수에서 모듈이 내려가면 proc파일과 디렉토리가 제거되도록 했습니다. 이때 remove_proc_entry 함수와 proc_create 함수를 사용했습니다.

blk-core.c에서 각 queue를 외부참조하여, seq_file API를 사용해 출력하였습니다. ptr_num1, 2도 외부참조하여 현재 queue가 어디까지 채워졌는지 알 수 있습니다.

○ 실행 방법에 대한 설명 및 캡쳐

1. 실행방법

: ext4_dir, f2fs_dir 두 개의 디렉토리를 만들어 f2fs_dir에는 f2fs 파일시스템을 마운트했습니다. 각각의 디렉토리에서 iozone을 통해 테스트를 실행해 비교했습니다. 결과 출력을 위해 커널모듈을 통해 생성된 proc file을 사용했습니다. 두 번의 테스트 후 각각의 결과를 해당하는 proc file을 통해 확인했습니다.

2. 쓰기 동작 시나리오

: -i 0 옵션과 -f 옵션을 통해 임시 파일에서 write/rewrite 에 대한 테스트를 수행한 결과 f2fs는 쓰기 동작을 비교하기 위한 로드 수가 충분하지 않았습니다. 쓰기 요청이 발생하면 바로바로 디스크에 write하는 ext4 방식과 달리, f2fs는 쓰기요청을 특정 단위로 모아서 한번에 처리하기 때문에 그런 것으로 판단하고 충분한 로드 수를 위해 같은 테스트를 여러번 실행해서 circular queue에 충분한 데이터가 들어가도록 했습니다. 최종적으로 저희가 사용한 옵션을 포함한 명령어는

./iozone -r 512b -s 100m -i 0 -f [파일디렉토리]

./iozone -r 8k -s 100m -i 0 -f [파일디렉토리]

이었습니다.

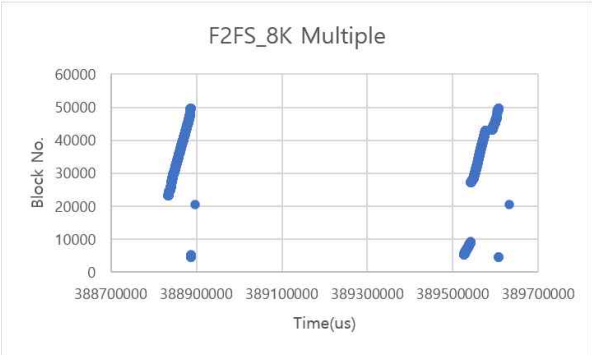
3. 캡쳐화면

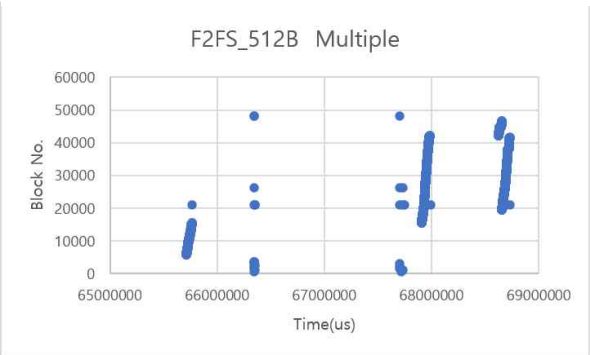
- ext4 실험결과

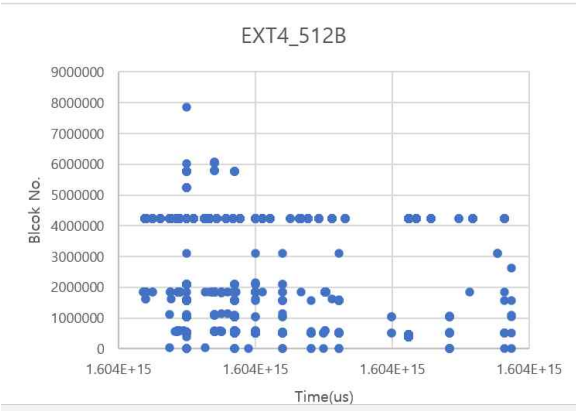
```
sungyoon@sungyoon-VirtualBox:~/lozone3_414/src/current$ sudo cat /proc/test3/ext4_proc
time : 1604112902376211 || FS_name : ext4 || block_no : 3713683
time : 1604112902376211 || FS_name : ext4 || block_no : 3713684
time : 1604112902376432 || FS_name : ext4 || block_no : 3713685
time : 1604112902377891 || FS_name : ext4 || block_no : 1158144
time : 1604112902378213 || FS_name : ext4 || block_no : 3713686
time : 1604112902378218 || FS_name : ext4 || block_no : 3713687
time : 1604112902378384 || FS_name : ext4 || block_no : 3713688
time : 1604112902379875 || FS_name : ext4 || block_no : 1
time : 1604112902379883 || FS_name : ext4 || block_no : 524363
time : 1604112902379886 || FS_name : ext4 || block_no : 736162
time : 1604112902379888 || FS_name : ext4 || block_no : 1572877
time : 1604112902379890 || FS_name : ext4 || block_no : 2097167
time : 1604112902393276 || FS_name : ext4 || block_no : 3569936
time : 1604112902396049 || FS_name : ext4 || block_no : 3713689
time : 1604112902396055 || FS_name : ext4 || block_no : 3713690
time : 1604112902396275 || FS_name : ext4 || block_no : 3713691
time : 1604112902447533 || FS_name : ext4 || block_no : 2830115
time : 1604112902447672 || FS_name : ext4 || block_no : 2830371
time : 1604112902447897 || FS_name : ext4 || block_no : 2830627
time : 1604112902448038 || FS_name : ext4 || block_no : 2830883
time : 1604112902448152 || FS_name : ext4 || block_no : 2831139
time : 1604112902448259 || FS_name : ext4 || block_no : 2831395
time : 1604112902448365 || FS_name : ext4 || block_no : 2831651
time : 1604112902448469 || FS_name : ext4 || block_no : 2831907
time : 1604112902448574 || FS_name : ext4 || block_no : 2832163
time : 1604112902448679 || FS_name : ext4 || block_no : 2832419
time : 1604112902448785 || FS_name : ext4 || block_no : 2832675
time : 1604112902448891 || FS_name : ext4 || block_no : 2832931
time : 1604112902449022 || FS_name : ext4 || block_no : 2833187
time : 1604112902449134 || FS_name : ext4 || block_no : 2833443
time : 1604112902449244 || FS_name : ext4 || block_no : 2833699
time : 1604112902449349 || FS_name : ext4 || block_no : 2833955
time : 1604112902449360 || FS_name : ext4 || block_no : 3569937
time : 1604112902489279 || FS_name : ext4 || block_no : 3713692
time : 1604112902489299 || FS_name : ext4 || block_no : 3713693
time : 1604112902489439 || FS_name : ext4 || block_no : 3713694
time : 1604112902546137 || FS_name : ext4 || block_no : 2834211
time : 1604112902546256 || FS_name : ext4 || block_no : 2897920
time : 1604112902546369 || FS_name : ext4 || block_no : 2898176
time : 1604112902546491 || FS_name : ext4 || block_no : 2898432
time : 1604112902546648 || FS_name : ext4 || block_no : 2898688
time : 1604112902546770 || FS_name : ext4 || block_no : 2898944
time : 1604112902546885 || FS_name : ext4 || block_no : 2899200
time : 1604112902546998 || FS_name : ext4 || block_no : 2899456
time : 1604112902547113 || FS_name : ext4 || block_no : 2899712
time : 1604112902547233 || FS_name : ext4 || block_no : 3436544
time : 1604112902547357 || FS_name : ext4 || block_no : 3436800
time : 1604112902547480 || FS_name : ext4 || block_no : 3437056
time : 1604112902547651 || FS_name : ext4 || block_no : 3437312
time : 1604112902548531 || FS_name : ext4 || block_no : 3437568
```

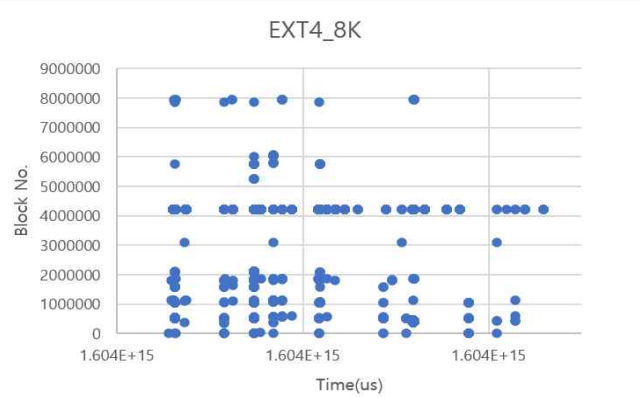
- f2fs 실험결과

```
sungyoon@sungyoon-VirtualBox:~/lozone3_414/src/current$ sudo cat /proc/f2fs_proc/f2fs_proc
time : 1604112576601558 || FS_name : f2fs || block_no : 27724
time : 1604112576601773 || FS_name : f2fs || block_no : 27755
time : 1604112576601894 || FS_name : f2fs || block_no : 27786
time : 1604112576602016 || FS_name : f2fs || block_no : 27817
time : 1604112576602137 || FS_name : f2fs || block_no : 27848
time : 1604112576602257 || FS_name : f2fs || block_no : 27879
time : 1604112576602377 || FS_name : f2fs || block_no : 27910
time : 1604112576602531 || FS_name : f2fs || block_no : 27941
time : 1604112576602654 || FS_name : f2fs || block_no : 27972
time : 1604112576602872 || FS_name : f2fs || block_no : 28003
time : 1604112576603176 || FS_name : f2fs || block_no : 28034
time : 1604112576603348 || FS_name : f2fs || block_no : 28065
time : 1604112576603492 || FS_name : f2fs || block_no : 28096
time : 1604112576603755 || FS_name : f2fs || block_no : 28127
time : 1604112576605751 || FS_name : f2fs || block_no : 28158
time : 1604112576605908 || FS_name : f2fs || block_no : 11264
time : 1604112576606051 || FS_name : f2fs || block_no : 11295
time : 1604112576606169 || FS_name : f2fs || block_no : 11326
time : 1604112576606319 || FS_name : f2fs || block_no : 11357
time : 1604112576606445 || FS_name : f2fs || block_no : 11388
time : 1604112576606564 || FS_name : f2fs || block_no : 11419
time : 1604112576606679 || FS_name : f2fs || block_no : 11450
time : 1604112576606798 || FS_name : f2fs || block_no : 11481
time : 1604112576606916 || FS_name : f2fs || block_no : 11512
time : 1604112576607153 || FS_name : f2fs || block_no : 11543
time : 1604112576607299 || FS_name : f2fs || block_no : 11574
time : 1604112576607424 || FS_name : f2fs || block_no : 11605
time : 1604112576607557 || FS_name : f2fs || block_no : 11636
time : 1604112576607680 || FS_name : f2fs || block_no : 11667
time : 1604112576607896 || FS_name : f2fs || block_no : 11698
time : 1604112576608016 || FS_name : f2fs || block_no : 11729
time : 1604112576608139 || FS_name : f2fs || block_no : 11760
time : 1604112576608279 || FS_name : f2fs || block_no : 11791
time : 1604112576608402 || FS_name : f2fs || block_no : 11822
time : 1604112576608520 || FS_name : f2fs || block_no : 11853
time : 1604112576608640 || FS_name : f2fs || block_no : 11884
time : 1604112576608808 || FS_name : f2fs || block_no : 11915
time : 1604112576608931 || FS_name : f2fs || block_no : 11946
time : 1604112576609050 || FS_name : f2fs || block_no : 11977
time : 1604112576609172 || FS_name : f2fs || block_no : 12008
time : 1604112576609349 || FS_name : f2fs || block_no : 12039
time : 1604112576609468 || FS_name : f2fs || block_no : 12070
time : 1604112576609585 || FS_name : f2fs || block_no : 12101
time : 1604112576609703 || FS_name : f2fs || block_no : 12132
time : 1604112576609819 || FS_name : f2fs || block_no : 12163
time : 1604112576609934 || FS_name : f2fs || block_no : 12194
time : 1604112576610051 || FS_name : f2fs || block_no : 12225
time : 1604112576610174 || FS_name : f2fs || block_no : 12256
time : 1604112576610331 || FS_name : f2fs || block_no : 12287
time : 1604112576610449 || FS_name : f2fs || block_no : 12318
```









○ 결과 그래프 및 분석

ext4의 경우, write 요청이 발생할 때 접근하는 block number가 random 하지만, f2fs는 write 요청이 발생하면 접근하는 block number 도 같이 증가하는 것을 확인할 수 있었습니다. f2fs는 시간이 지나면 디스크 공간이 부족해지므로 계속해서 freespace management를 해줘야합니다. sequential한 write가 진행되는 도중 sequential하지 않은 임의의 블록에 접근하는 것은 f2fs가 freespace management를 수행하고 있는 결과라고 판단했습니다.

추가로, 각 8K와 512B의 의미는 test하는 record size의 크기를 의미합니다. -r 옵션으로 정해줄 수 있습니다. 저희는 같은 파일 시스템 안에서 record size가 어떤 의미가 있는지 의문이 들어 직접 비교를 해 보았습니다. F2FS의 경우는 8K와 512B 사이에 차이가 없었습니다. 그러나 EXT4의 경우 512B의 경우가 iozone test를 한 번 시행했을 경우 기록되는 block의 수가 8K보다 훨씬 많았음을 발견했습니다. 이는, 우리가 이미 수업에서 배웠던 것처럼 LFS가 write를 할 때는 특정 단위로 모아서 한번에 처리하는 것과 관련 있다고 추측했습니다.

○ 과제 수행 시 어려웠던 부분과 해결 방법

1. proc 출력

proc에 원하는 내용을 출력하는데 난항이 있었습니다. 출력 방법에 대해 조사하는 중, 실제 proc file들이 출력을 어떻게 하는지 안다면 출력에 관한 실마리를 얻을 수 있을 것이라 생각해 /proc/stat.c의 코드를 읽어보았습니다. 이미 만들어져 있는 proc 파일들은 seq_file API를 사용해 출력을 한다는 것을 깨달았고, 그 API를 사용하여 문제를 해결하였습니다.

2. timestamp

- wirte 요청이 발생할 때의 timestamp를 저장하기 위해 현재 시간에 대한 정보를 찾기 위해 시간이 걸렸습니다. timestamp 또한, proc file 자체가 컴퓨터 시스템에 대한 정보를 출력해준다는 것에 착안하여 어떤 proc file은 현재 시간을 출력해 줄 것이란 추측을 하였습니다. 이 추측을 토대로 proc file을 조사하던 중 time_keeping.h에서 현재 시간을 출력해 주는 함수를 찾아 활용하였습니다.

3. redefined

- q_entry 구조체를 처음에는 blk-core.c 에 선언을 했는데, 이렇게 되면 모듈파일에서 q_entry 구조체에 대한 정보가 없기 때문에 모듈파일에도 q_entry를 선언해서 사용했습니다. 그러나 그로인해 redefined 에러가 발생했습니다. 이를 해결하기 위해 q_entry만을 따로 선언하는 header 파일을 생성해 include함으로써 redefined 문제를 해결했습니다.

4. unknown symbol

- 커널 모듈을 작성해서 Makefile과 함께 컴파일 했지만 모듈 적재가 안되는 문제가 있었습니다. unknown symbol이라는 메시지를 보고 proc file system 의 kallsyms를 확인하니 symbol이 제대로 export 되지 않았음을 확인했습니다. 커널 컴파일 후 reboot를 하지 않아 symbol이 커널에 제대로 생성되지 않은 아주 간단한 문제였습니다.