2017년 2학기 시스템 프로그래밍

1차 과제

– LFS(Log-structured File System) Profiling –

[SP 12]

2015410012 김현섭

2015410017 채호경

제출일자: 2017. 11. 03 금요일

프리 데이 사용일수: 2일

목차

1. 역할 배분

기본적인 코드에 대한 분석은 같이 진행했으나, 코드 작성의 경우는 두 명이 동시에 진행하기 힘들다고 판단해 김현섭이 주로 작성하였다. 채호경은 섹터 접근 시간 출력에 있어서 do_gettimeofday 함수의 적용과 결과를 csv 파일로 저장하는 방법을 찾아서 적용하였고, 보고서 작성에 필요한 자료 조사 및 결과 그래프 분석을 담당하였다. 이외의 전반적인 코드 작성은 김현섭이 담당하였다.

2. 개발 환경

가상머신 : Oracle VM VirtualBox

운영체제 : Ubuntu 16.04 LTS

커널 : linux-4.4.40

파일시스템 : Nilfs2

3. 배경 지식

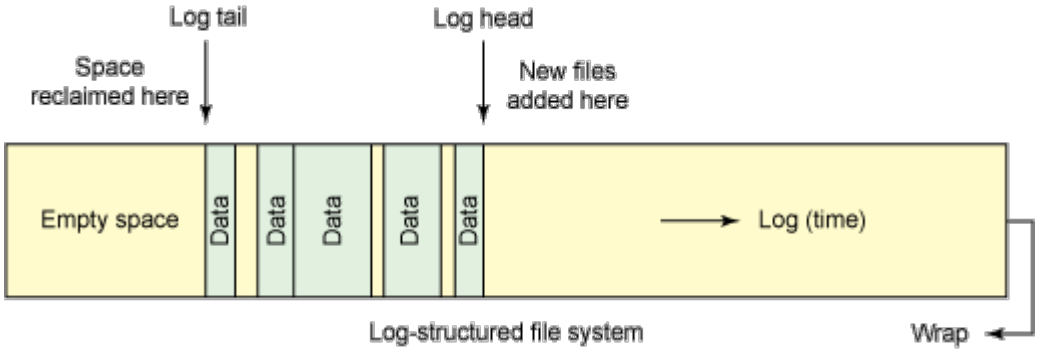
Nilfs2는 Linux상에 구현된, continuous snapshotting을 제공하는 log structured file system이다. 기존의 다른 일반적인 file system과 비교하여 LFS의 특징을 조사하였다. 기본적으로 Linux 환경에서 ext4와 같은 일반적인 file system은 파일을 저장할 때 생성 시간, 수정 시간, 접근 권한 등 다양한 정보를 포함하는 메타데이터가 저장되어있는 inode는 실제 파일의 데이터가 저장된 블록들의 위치 정보를 가지고 있다. inode를 식별하기 위한 고유 번호(inode number)와 파일명이 함께 묶여서 dentry라는 구조체가 되고, 이를 통해 사용자는 파일에 접근한다. 리눅스 터미널에서 ls -i를 입력하면 inode의 번호와 파일 이름이 맵핑되어 있는 것을 볼 수 있다.

이렇게 파일의 메타데이터를 가지고 있는 inode들은 물리 디스크 상에 흩어져 저장되기 때문에 어느 위치에 저장되어 있는지 tracking이 필요하다. 따라서 super block이라는 구조체가 inode들의 위치 정보를 가지고 있고, ext4에서 특정 파일에 write를 수행하면, 물리 디스크 상에서 서로 떨어져 있는 여러 블록에 분산되어 write 작업이 발생할 것이다. 이렇게 흩어져 있는 데이터들은 super block을 통해 해당 파일의 inode의 위치를 알아내고, 그 inode를 참조하여 실제 데이터들이 저장되어 있는 곳을 찾는 과정을 통해서 읽고 수정할 수 있다. 반면 이번 과제의 Nilfs2와 같은 Log Structured file system은 이와 차이점이 있다.

[LFS]

Log-structured File System

첫 번째 Log-structured file system은 John Ousterhout와 Fred Douglis가 1988년에 제안했으며 그 후 1992년에 Sprite 운영 체제에서 구현되었다. 이름에서 알 수 있듯이 Log-structured file system에서는 file system을 새 데이터와 file system 메타데이터를 로그의 헤드에 쓴 순환 로그로 여기며 여유 공간은 끝에서부터 사용된다. 이렇게 하면 로그에 데이터가 두 번 이상 표시될 수도 있지만 로그는 시간에 따라 정렬되기 때문에 가장 최근의 데이터가 활성 데이터로 표시된다. 로그에 다수의 데이터 사본이 있으면 몇 가지 중요한 장점이 생기며 이러한 사항은 아래에서 자세히 설명한다.



〈Log-structured file system의 간단한 구조〉

Log-structured 방식은 장점이라기보다는 아키텍처적인 개념이라고 할 수 있으며 이러한 방식은 몇 가지 독특한 장점을 제공한다. 한 가지 중요한 장점으로는 시스템이 파손되었을 때 복구하는 기능을 들 수 있으며 Log-structured 방식을 사용하면 이 기능이 더 단순해진다.

또 다른 장점은 기본 스토리지 시스템을 사용하여 성능을 개선할 수 있다는 점이다. 순차적으로 디스크에 쓰는 동작은 무작위로 I/O에 쓰는 동작보다 훨씬 더 빠르다. 쓰기 동작은 모두 순차적으로 행해지기 때문에 찾기에 대한 부담이 제거되고 결과적으로 디스크 I/O 속도가 빨라져 file system의 속도가 개선된다.

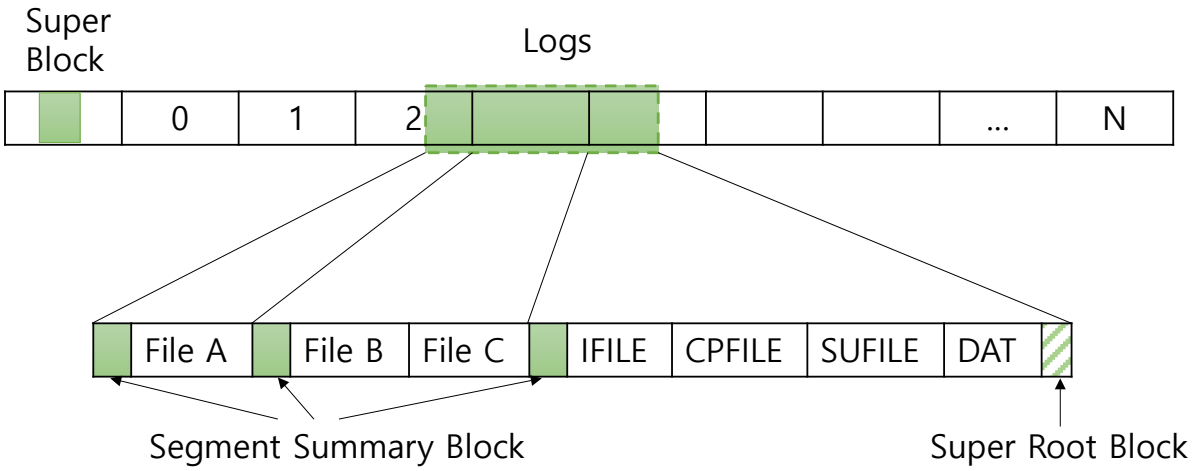
하나의 파일에 write를 할 때 여러 블록들이 떨어진 곳에 위치해 있어서 디스크 접근 시간이 오래 걸린다는 문제점을 보완하기 위해 write 작업을 연속된 공간에 순차적으로 진행한다. 즉, 떨어져 있는 공간이 아니라 연속된 디스크 블록에 실제 데이터들을 쓰고, 그에 대한 메타데이터도 역시 따로 특정한 곳에 저장을 하는 것이 아니라 실제 데이터와 함께 연속적으로 저장을 한다. 새로운 파일 생성하는 것뿐만 아니라, 기존의 파일을 수정할 때도 ext4처럼 inode를 통해 실제 데이터 블록들의 위치 정보를 참조하여 직접 수정하는 것이 아니라, 그냥 마지막으로 쓰기를 마친 지점부터 순차적으로 새로 쓰기를 진행한다. 즉, 계속 append 작업만 수행하는 것이다.

그런데, 이는 이전 데이터를 계속 디스크 상에 남겨 놓는다는 점에서 필연적으로 디스크 공간을 낭비할 수 밖에 없다. 따라서 지속적으로 garbage collection 과정을 통해 디스크 공간의 여유분을 확보해주어야 한다. 그리고 또한, 파일 읽기를 수행할 때의 문제점이 있다. 쓰기는 순차적으로 진행하기 때문에 매우 간편하고 효율적이지만, 파일 읽기의 경우에는 무차별적인 append 작업에 의해 어디에 파일이 저장되어 있는지를 알 수 없기 때문이다. ext4의 super block 처럼 inode들의 위치 정보를 저장해둘 것이 필요하다. 이러한 문제를 해결하기 위해, inode map이라는 것을 메모리에 caching해서 수많은 inode의 copy들 중에 어떤 것이 가장 최근의 유효한 inode인지 그 정보들을 저장해둔다.

[Nilfs2]

Nilfs2는 Log structure File System의 하나로 본래 하드디스크를 대상으로 개발되었다. Log structure File System은 덮어쓰기를 하지 않기 때문에 변경 되기 전의 상태가 저장장치에 그대로 남는데, 이런 특성을 이용하여 스냅샷(Snapshot) 기능을 제공하는 것이 Nilfs2의 개발 목적이다. 본래 플래시 메모리를 대상으로 설계되지 않았지만 Log structure File System이기 때문에 플래시 메모리에서 사용하기에 유리한 Log structure File System의 특성은 그대로 가지고 있다. 덮어쓰기를 하지 않고 세그먼트 내에서 순차적으로 기록한다는 점이 바로 그것이다.

Nilfs2는 순수한 Log structure File System에 속한다. 즉 메타데이터와 데이터 모두 로그에 기록한다. 또한 메타데이터를 파일로 처리하는 것이 특징이다. 즉 메타데이터를 저장하는 데에 별도의 자료 구조(이를테면 테이블)를 사용하는 것이 아니라 일반 데이터처럼 메타데이터 또한 파일 안에 담는다.



[Nilfs2의 저장장치 상 레이아웃]

1) Inode file (ifile) —— Stores on-disk inodes

2) Checkpoint file (cpfile) —— Stores checkpoints

3) Segment usage file (sufile) —— Stores allocation state of segments

4) Data address translation file (DAT) —— Maps virtual block numbers to usual block numbers. This file serves to make on-disk blocks relocatable.

위 그림은 Nilfs2의 저장장치 상의 레이아웃을 나타낸 구조도이다. Log structure File System이기 때문에 LBA 주소 공간을 세그먼트 단위로 나누어서 관리하는데 그림에서 '0, 1, 2, ...'로 표시된 것들이 세그먼트 번호를 나타낸다. 세그먼트는 로그의 컨테이너이다. 각 로그는 요약 정보 블록, 페이 로드 블록 및 옵션인 Super Root block(SR)으로 구성된다. 저장 공간 전체가 로그 영역이며 변경한 내용이 차례로 추가된다. 로그 기록할 때마다 세그먼트 요약 블록(Segment Summary Block)이 제일 앞에 기록되어 각 로그를 구분해준다. 그림에서 'File A' 식으로 표시된 것은 파일 A에 대한 변경사항을 기록한 블록들이라는 것을 나타낸다. 그림에서 IFILE, CPFILE, SUFILE, DAT는 NILFS2에서 사용하는 메타데이터 파일들을 나타낸다. 파일로 처리되기 때문에 이것들 또한 변경된 부분들만 로그에 포함된다. 메타데이터 파일에 대한 변경 내역은 sync를 해야 할 때 비로소 로그에 기록되고 제일 마지막에 'Super Root Block'이 추가된다.

I/O 성능 측면에서는 불리한 편인데, 기본적으로 스냅샷을 기능을 제공하기 위해서 추가적으로 관리해야 하는 메타데이터가 많기 때문이다. 위 그림에 나타난 메타데이터 파일 항목 중에서 DAT 같은 경우는 통상적인 Log structure File System에는 아예 없는 요소이며 체크포인트에 해당되는 CPFILE의 구조도 다른 Log structure File System과 비교하면 상대적으로 복잡한 편이다.

Nilfs2에서 주목할 만한 부분은 연속해서 스냅샷을 할 수 있는 기술이다. Nilfs2는 로그 형태로 구조화되어 있기 때문에 새 데이터는 로그의 헤드에 쓰여지며 기존 데이터는 garbage collection되지 않는 한 계속 존재한다. 기존 데이터가 계속 존재하기 때문에 원하는 시점으로 돌아가서 해당 file system의 에포크(epoch)를 조사할 수 있다. 이러한 에포크는 Nilfs2에서 체크포인트라고 불리며 file system의 핵심 부분이 된다. Nilfs2에서는 특정 시간 또는 file system에 변화가 생길 때마다 이러한 체크포인트가 작성되지만 사용자가 강제로 체크포인트를 작성할 수도 있다. 사용자들은 일련의 저장된 checkpoint들 중 특정한 버전

을 선택하여, 이를 snapshot으로 바꾸어 보존할 수 있다. 볼륨이 가득차기 전까지 만들 수 있는 snapshot의 개수에는 한계가 없다. 각각의 snapshot은 read-only file system으로서 mount와 동시에 writable mount 역시 가능하다.

nilfs-utils package에는 사용자층 tool들이 포함되어있고, 아래와 같다.

lscp - list NILFS2 checkpoints

lssu - list usage state of NILFS2 segments

nilfs_cleanerd.conf - nilfs_cleanerd(8) configuration file

chcp - change mode of NILFS2 checkpoints

dumpseg - print segment information of NILFS2

mkcp - make a NILFS2 checkpoint

mkfs.nilfs2 - create a NILFS2 filesystem

nilfs-clean - run garbage collector on NILFS file system

nilfs-resize - resize NILFS file system volume size

nilfs-tune - adjust tunable file system parameters on NILFS file system

NILFS - the new implementation of a log-structured file system

nilfs_cleanerd - NILFS2 garbage collector

rmcp - remove NILFS2 checkpoints

umount.nilfs2 - unmount NILFS2 file systems

4. 작성 부분 설명

blk-core.c

```
//HyunsubKim 2017-11-02 Start ================================================
#define SP_BUFFER_SIZE 1000
#include <linux/time.h>
//HyunsubKim 2017-11-02 End ==================================================
```

먼저 버퍼 사이즈를 #define을 사용하여 선언해 주었고 do_gettimeofday를 사용할 time.h를 include 시켜 주었다.

```
//HyunsubKim 2017-11-02 Start ================================================
unsigned long long sp_sector_arr[SP_BUFFER_SIZE];
unsigned long long sp_time_arr[SP_BUFFER_SIZE];
int sp_arr_index = 0;
struct timeval sp_timeval;

EXPORT_SYMBOL(sp_sector_arr);
EXPORT_SYMBOL(sp_arr_index);
EXPORT_SYMBOL(sp_time_arr);
//HyunsubKim 2017-11-02 End ==================================================
```

섹터 번호를 저장할 배열, 시간을 저장할 배열, 배열의 인덱스, 그리고 dogettimeofday에 사용할 timeval 을 선언하고 이 중 섹터 번호 배열과 시간 배열과 배열의 인덱스를 EXPORT_SYMBOL을 사용하여 hw1.c 에서도 사용 할 수 있게 했다.

```
//HyunsubKim 2017-11-02 Start ====================================
if(bio->bi_iter.bi_sector != 0) {
    int is_nilfs = 0;

    if(bio->bi_bdev != NULL)
        if(bio->bi_bdev->bd_super != NULL)
            if(bio->bi_bdev->bd_super->s_type != NULL)
                if(bio->bi_bdev->bd_super->s_type->name != NULL)
                    is_nilfs = strcmp(bio->bi_bdev->bd_super->s_type->name, "nilfs2") == 0;

    if(is_nilfs) {
        do_gettimeofday(&sp_timeval);

        sp_sector_arr[sp_arr_index] = (unsigned long long) bio->bi_iter.bi_sector;
        sp_time_arr[sp_arr_index] = (unsigned long long) (sp_timeval.tv_sec * 1000000 + sp_timeval.tv_usec);
        sp_arr_index = (sp_arr_index + 1) % SP_BUFFER_SIZE;
    }
}
//HyunsubKim 2017-11-02 End ======================================
```

먼저 섹터 번호가 valid 한지 확인 한 후 이 block io 구조체의 파일 시스템 이름이 nilfs2와 같은지 확인 한 후 같으면 is_nilfs 변수에 true(1)을 집어넣는다.

만약에 nilfs2 파일 시스템의 block io 이면 섹터 번호와 현재 시간을 배열에 집어넣고 배열의 인덱스를 1 증가 시킨다.

segbuf.c

```
//HyunsubKim 2017-11-02 Start ================================================
if(bio->bi_bdev != NULL)
    bio->bi_bdev->bd_super = segbuf->sb_super;
//HyunsubKim 2017-11-02 End ==================================================
```

nilfs2는 block io를 수행할 때, submit_bio 함수가 호출 되기 전까지 자신만의 구조체를 사용하므로 블록마다 파일 시스템 정보를 가지도록 했다.

5. 실행 방법 설명 및 실행 결과 캡쳐 화면

1) blk-core.c를 $리눅스경로/block/에 넣고, segbuf.c는 $리눅스경로/fs/nilfs2/에 넣는다.

2) 커널을 컴파일하고 설치한다.

3) hw1.c를 make하고, 나온 hw1.ko 모듈을 insmod 명령어를 통해 커널에 삽입한다.

4) nilfs 파일시스템을 쓰는 가상디스크를 만들어 포맷한 후

I/O device로 등록하여 마운트한다.

5) 마운트한 디렉토리에서 iozone test를 실행한다.

6) /proc/myproc/ 경로로 접근하여 echo 아무글씨 〉 myproc 명령어를 수행한다.

7) 출력된 값은 /tmp/에 생성된 result.csv 파일 또는 dmesg 명령어로 확인 가능하다.

버퍼 인덱스, 시간, 섹터 넘버 순으로 출력된다.

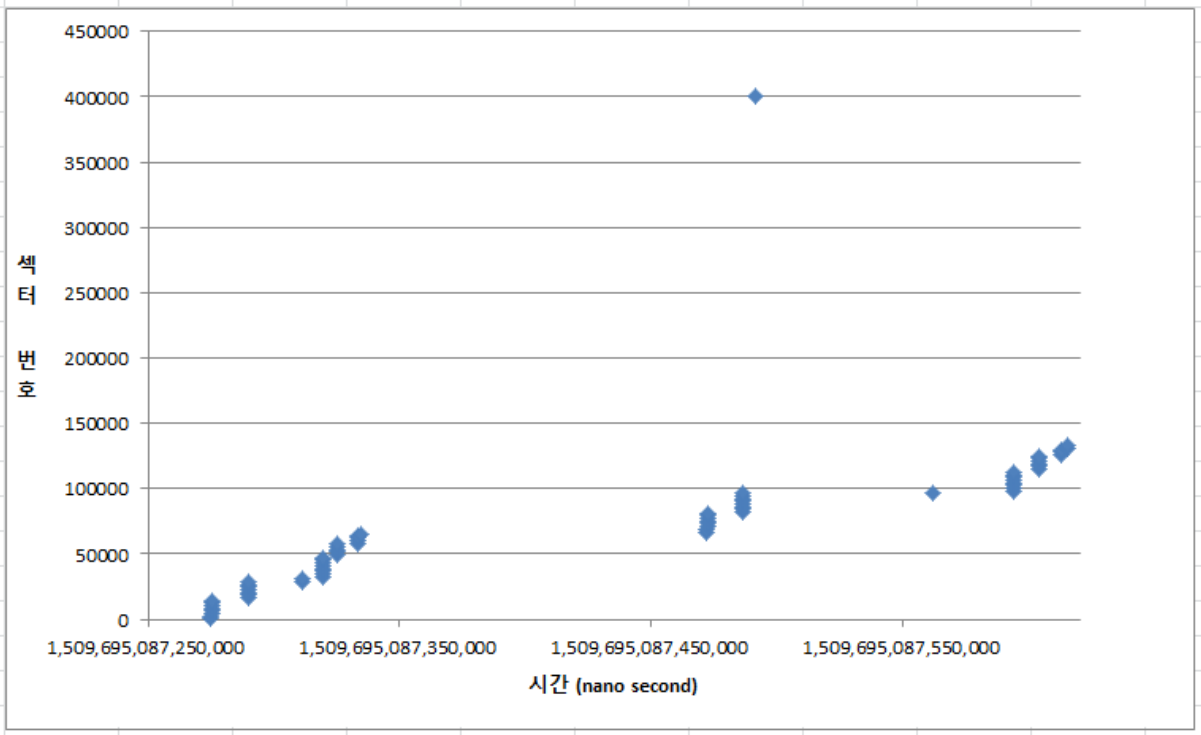
```
1509700251146705,60552
1509700251147209,62600
1509700251147668,64648
1509700251148458,65536
1509700251281126,66840
1509700251281716,68888
1509700251282294,70936
1509700251282791,72984
1509700251283478,75032
1509700251283958,77080
1509700251284419,79128
1509700251284920,81176
1509700251300061,81920
1509700251300660,83968
1509700251301159,86016
1509700251301831,88064
1509700251302283,90112
1509700251302755,92160
1509700251303116,94208
1509700251303480,96256
1509700251304203,399992
1509700251371226,96568
1509700251396388,98304
1509700251396871,100352
1509700251397310,102400
1509700251397832,104448
1509700251398212,106496
1509700251398607,108544
1509700251398988,110592
1509700251399364,112640
1509700251407662,114688
1509700251408041,116736
1509700251408444,118784
1509700251408819,120832
1509700251409194,122880
1509700251409632,124928
1509700251415606,125760
1509700251416003,127808
1509700251416392,129856
1509700251418760,131072
1509700251419156,133120
1509700256481424,134152
root@noverish-VirtualBox:~#
```

Csv file

```
[    89.076303] i : 59
[    89.076306] sp_time_arr[i] : 1509700251398988
[    89.076309] sp_sector_arr[i] : 110592
[    89.076310] i : 60
[    89.076313] sp_time_arr[i] : 1509700251399364
[    89.076316] sp_sector_arr[i] : 112640
[    89.076316] i : 61
[    89.076319] sp_time_arr[i] : 1509700251407662
[    89.076322] sp_sector_arr[i] : 114688
[    89.076323] i : 62
[    89.076326] sp_time_arr[i] : 1509700251408041
[    89.076329] sp_sector_arr[i] : 116736
[    89.076330] i : 63
[    89.076332] sp_time_arr[i] : 1509700251408444
[    89.076335] sp_sector_arr[i] : 118784
[    89.076336] i : 64
[    89.076339] sp_time_arr[i] : 1509700251408819
[    89.076342] sp_sector_arr[i] : 120832
[    89.076343] i : 65
[    89.076345] sp_time_arr[i] : 1509700251409194
[    89.076367] sp_sector_arr[i] : 122880
[    89.076368] i : 66
[    89.076371] sp_time_arr[i] : 1509700251409632
[    89.076374] sp_sector_arr[i] : 124928
[    89.076375] i : 67
[    89.076377] sp_time_arr[i] : 1509700251415606
[    89.076380] sp_sector_arr[i] : 125760
[    89.076381] i : 68
[    89.076384] sp_time_arr[i] : 1509700251416003
[    89.076387] sp_sector_arr[i] : 127808
[    89.076388] i : 69
[    89.076390] sp_time_arr[i] : 1509700251416392
[    89.076393] sp_sector_arr[i] : 129856
[    89.076394] i : 70
[    89.076397] sp_time_arr[i] : 1509700251418760
[    89.076400] sp_sector_arr[i] : 131072
[    89.076401] i : 71
[    89.076403] sp_time_arr[i] : 1509700251419156
[    89.076406] sp_sector_arr[i] : 133120
[    89.076407] i : 72
[    89.076410] sp_time_arr[i] : 1509700256481424
[    89.076413] sp_sector_arr[i] : 134152
root@noverish-VirtualBox:~#
```

dmesg 결과 화면

6. 결과 그래프 및 그에 대한 설명



그래프를 보면 Log structured File System의 특징이 나타나게 되는데, 시간의 흐름에 따라 연속된 블록들에 접근하는 것을 알 수 있었고, 이는 곧 Write를 수행함에 있어서 메타데이터와 data block을 구분하여 별도의 공간에 저장하는 것이 아니라 연속된 공간에 순차적으로 접근한다는 것을 확인하였다. Kernel에 write에 대한 시스템 콜을 보냈으나 kernel이 바쁜 관계로 write가 잠시 일어나지 않았다고 추측할 수 있다.

****수업시간에 배운 FFS(Fast File System)에 동일한 실험을 수행한다면, 어떤 결과가 나올지 설명 및 nilfs2의 쓰기 패턴과 비교 분석하기**

FFS의 개선점으로는 Throughput을 높이기 위해 블록 사이즈를 4KB로 설정하였고, 커진 블록 사이즈로 인해 발생한 공간 낭비 문제를 Fragmentation으로 해결한다. 그리고 Cylinder 그룹을 정의 하여 한 Cylinder 그룹에 superblock, indoe, file들을 배치하는데 이때, 같은 디렉토리에 속하는 파일과 inode를 같은 cylinder 그룹에 할당한다. 즉, 파일에 대한 data block이 되도록 같은 cylinder에 포함되므로 접근하는 블록 넘버들이 LFS와 같이 항상 순차적으로 증가하지는 않겠지만, 일정 시간 동안 사용되는 블록 넘버들의 폭이 지나치게 넓지는 않을 것임을 예측할 수 있다. 비록 iozone 벤치마크를 FFS 상에서 실행해서 확인하지는 않았기 때문에 구체적인 그래프 양상을 하나로 특정지을 수는 없으나, 대략적인 특징으로는 일정 시간에 대해 인접한 블록들에 있어서 반복적인 접근이 있을 것이고, 그러한 인접해있는 블록들의 묶음이 벤치마크에 의해 사용되는 디렉토리들의 수만큼 분포할 것이다.

즉, FFS의 경우는 Write 과정에 있어서 공간에 대한 locality를 유지하는 방식이고, Nilfs2는 시간에 대한 locality를 활용하는 방식이라는 특징을 그래프로부터 유추할 수 있다.

7. 과제 수행 시 어려웠던 부분과 해결 방법

파일이 써질 때 어떤 파일 시스템이 써지는지 확인 하지 않고 일단 block io 의 valid함만 가지고 판단해서 버퍼에 넣었다. 그렇게 하고 나서 buffer를 확인해 보니 결과 값이 들쭉날쭉 하고 예상하지 못한 결과가 나왔다. 디버깅을 통해 알아보니 buffer의 거의 대부분의 block io 들이 모두 Ext4에서 사용 된 것을 알았다. 처음에는 이해 할 수 없어서 다른 사람에게 물어보고 검색을 해보니 '저널링 파일 시스템' 이라는 것이 주기적으로 파일 시스템을 사용해서 buffer에 들어간 ext4의 block io가 전부 저널링 파일 시스템에서 수행한 것이라는 것을 알게 되었다. 유저인 내가 아무런 작업도 안 하면 당연히 파일 시스템에 아무런 시스템 콜이 생기지 않을 거라 착각했던 것이다. Buffer에 섹터 번호와 시간을 집어 넣을 때 nilfs 파일 시스템에서 수행하는 block io 인지를 확인하고 buffer에 넣었더니 올바른 값이 나왔다.