2014211491 전재형

1lantine1@naver.com

요약

[매력적인 요약문으로 독자의 시선을 끌어 보세요. 일반적으로 요약은 문서의 내용을 간략하게 정리한 것입니다.   
내용을 추가하려면 여기를 클릭하고 입력하세요.]

제목 미정

[문서 부제]

목차

[1. **기본 실험 환경 구성** 2](#_Toc39064686)

[**1.1 SSD 연결 확인 및 파티션 설정** 2](#_Toc39064687)

[**1.2 File System 포맷 및 Mount 하기** 3](#_Toc39064688)

[**1.3 Fio 벤치마크 설치** 4](#_Toc39064689)

[**1.4 XFS 파일 시스템 설치** 4](#_Toc39064690)

[**1.5 sudo 명령 암호 없이 이용하기** 5](#_Toc39064691)

[**1.6 shell script 작성하기** 6](#_Toc39064692)

[**1.7 기본 사전 지식 알아보기** 7](#_Toc39064693)

[**1.7.1 Fio benchmark Options** 7](#_Toc39064694)

[**1.7.2 XFS Options** 9](#_Toc39064695)

[**2. 실험** 11](#_Toc39064696)

[**2.1 Fio benchmark를 통한 Ext4와 XFS 파일시스템 성능비교** 11](#_Toc39064697)

[**2.1.1 Ext4파일시스템에서 bs 변화에 따른 IOPS값과 bw값** 11](#_Toc39064698)

[**2.1.2 XFS 파일 시스템에서 bs 변화에 따른 bw값과 IOPS값** 12](#_Toc39064699)

[**2.1.3 실험결과: 두 파일시스템에서 IOPS값과 bw값 비교** 13](#_Toc39064700)

[**2.2 fsync 옵션을 통한 파일시스템간 성능 비교** 14](#_Toc39064701)

[**2.2.1 두 파일 시스템에서의 결과값** 14](#_Toc39064702)

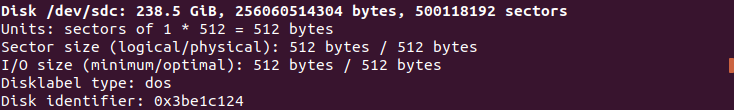
[**2.2.2 두 파일시스템의 결과값 비교** 15](#_Toc39064703)

[**2.2.3 실험결과** 19](#_Toc39064704)

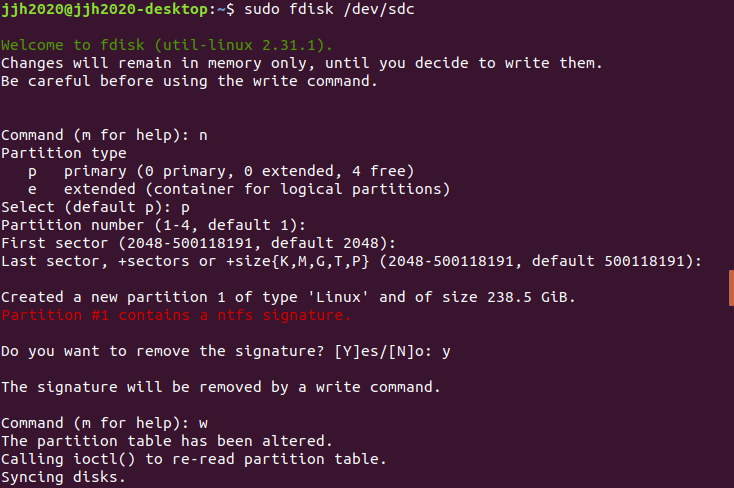
# 1. **기본 실험 환경 구성**

## **1.1 SSD 연결 확인 및 파티션 설정**

sudo fdisk –l로 연결된 ssd 확인



sudo fdisk /dev/sdc로 파티션 설정

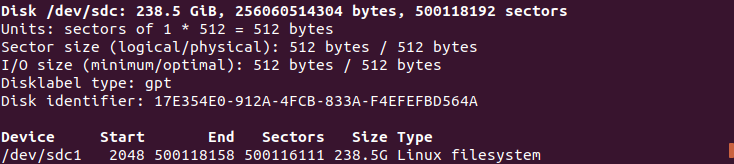


n을 입력하여 파티션 생성, p를 입력하여 기본 파티션 타입 설정

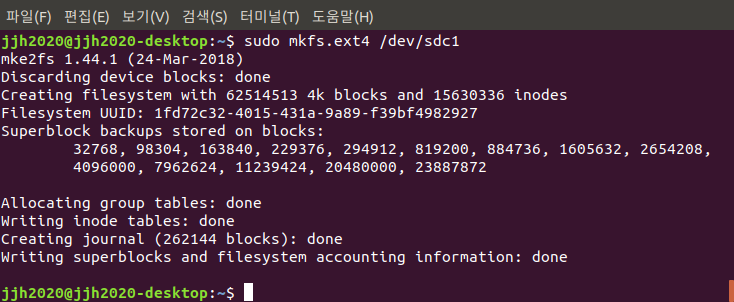
Enter 입력하여 default 1로 파티션 생성, 기존의 파일시스템을 지우겠냐고 물어보면 y,

w로 저장

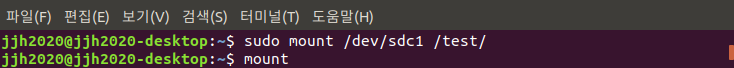
## **1.2 File System 포맷 및 Mount 하기**



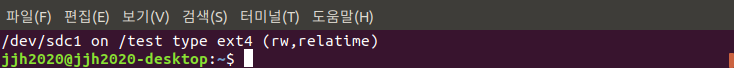
sudo fdisk –l로 정상적으로 설치된 것을 확인



sudo mkfs.ext4 /dev/sdc1으로 sdc1을 ext4 파일시스템으로 포맷

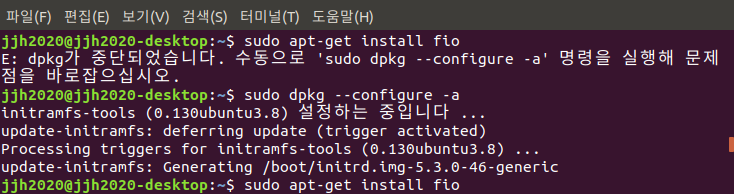


sudo mount /dev/sdc1 /test 를 입력하여 sdc1을 test폴더에 mount



mount 명령어로 mount가 된 것을 확인

## **1.3 Fio 벤치마크 설치**



sudo apt-get install fio 명령어로 fio 벤치마크 설치

만약 dpkg(데비안 패키지 관리시스템)에 문제가 있다면 수동으로 재설정

## **1.4 XFS 파일 시스템 설치**

C:\Users\user\AppData\Local\Microsoft\Windows\INetCache\Content.Word\스크린샷, 2020-04-15 12-13-57.png

Xfs 파일 시스템을 위의 명령어로 설치할 수 있다.

## **1.5 sudo 명령 암호 없이 이용하기**

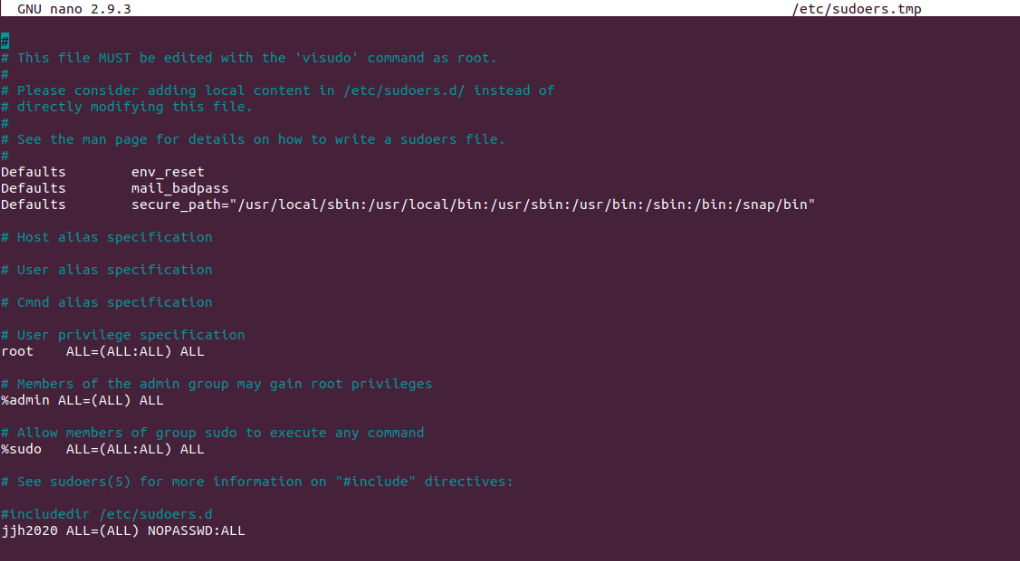
sudo는 superuser do의 약자로 현재 계정에서 root 권한을 이용하여 명령어를 실행할 때 사용한다.

실험을 할 때 터미널에서 일정 시간이 지나서 sudo 명령을 치면 암호를 입력하라고 하는데, shell script를 이용해서 실험을 돌리거나 실험 중 반복하여 암호를 입력하는 것을 피하기 위해 임의의 값을 추가하여 이를 피할 수 있다.

터미널에서 sudo su를 입력하고 암호를 입력하여 root 권한을 얻는다



그 후 visudo를 입력하여 Sudoers.tmp 파일을 연다.



맨 밑의 줄에 코드 (userid) ALL=(ALL) NOPASSWD:ALL를 입력한다.

Ctrl+O를 눌러서 저장하고 sudoers.tmp에서 .tmp를 빼고 sudoers로 저장하고 파일내용의 변경이나 Overwrite하겠냐고 묻는 말이 나오면 Y를 눌러준다.

Reboot 명령어를 입력해서 재시작 후 터미널에서 sudo 명령어를 사용하여 암호를 입력하는 것이 뜨는지 확인한다.

이제 여기서 암호를 입력하라고 뜨지 않는다면 성공.

암호를 입력하라고 나오거나 sudoers.tmp 에서 문법 오류가 발생한다면 다시 시도한다.

## **1.6 shell script 작성하기**

Fio 벤치마크를 이용하여 실험을 돌리다보면 time\_based 옵션으로 runtime 시간을 지정하여 반복하여 실험을 돌릴 수 있는데, 일정 시간 마다 반복하여 터미널에 일련의 과정을 반복해서 하는 번거로운 과정을 피하기 위해 미리 shell script에 명령어들을 입력해두고 편리하게 실험을 돌릴 수 있다.

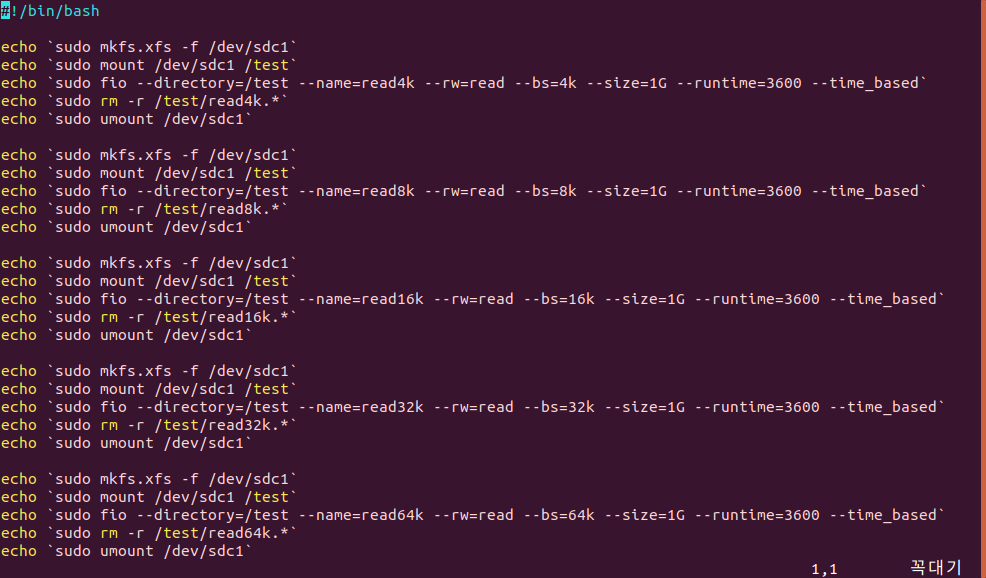
touch test.sh 명령어로 batch파일 생성

vim test.sh 명령어로 텍스트 에디터 열기

텍스트를 입력할 때에는 i를 눌러서 편집모드로 들어가고, Esc를 누르면 다시 명령모드로 돌아올 수 있다.

첫 번째 줄에 기본 쉘을 뜻하는 #!/bin/sh나 #!/bin/bash를 입력한다.

/bin/sh는 /bin/bash를 호출하기에 결과가 같다.



그 뒤로는 echo `명령어`를 입력하여 사용할 명령어들을 적어주고 Esc를 눌러 명령모드로 돌아온 후 :wq 를 입력하면 저장하고 vim을 종료한다.

편집한 test.sh를 실행할 때에는 chmod 명령어를 통해 파일에 권한을 부여해줘야 한다.

chmod u+x test.sh 를 통해서 user의 실행 권한을 부여한다.

./test.sh 명령을 입력하면 test.sh shell script가 작동하게 된다.

## **1.7 기본 사전 지식 알아보기**

### **1.7.1 Fio benchmark Options**

FIO는 Flexible IO test의 약자로 종합적인 오픈소스 벤치마크 툴이다.

유저들이 다양하게 옵션들을 정하여 IO 작업들을 발생시킬 수 있다.

장점: 일괄처리를 할 수 있고, 아주 넓은 parameter들의 집합을 가지고 있다.

IOMeter와 다르게 계속 개발 중이다. 무료로 이요할 수 있다.

단점: CLI(Command-Line Interface)만 지원한다.

Fio benchmark option들을 정해주기 전에 기본적인 설정 값들을 지정해줘야한다.

I/O type

Block size

I/O size

I/O engine

I/O depth

Target file/ device

block size : default value size가 4k, 읽기 쓰기에 쓰일 수 있음

random workloads에서 128k가 종종 쓰임

이 값을 변경하여 가능한한 실제 현장에서 쓰인 것처럼 사용해볼 수 있음

만약 서버가 4k block size로 99퍼의 시간을 다룬다면(차지한다면), 1mb block size로 왜 test를 해보나 <-?

ioengine

by default, fio는 sync io engine을 사용하여 테스트를 돌림

엔진은 변경할 수 있음. 다른 사용할 수 있는 많은 옵션들이 있지만,

커널에서 지원을 한다면 리눅스에서 가장 일반적인 옵션은 sync와 libaio

iodepth

이 옵션은 테스트동안 파일에 요청과 함께 두들기는걸(연속적인 접근?) 계속하면서 io unit의 총량을 규정한다.

default sync ioengine을 사용한다면, io depth의 값이 1을 넘어서는건 효과를 가지지 못한다.

libaro 같은 engine으로 변경해도 OS가 iodepth의 Maximum을 제한하고 specified value를 무시한다.

이것 때문에, io depth의 값을 1로 시작하고 값을 올려서 , 예를 들면 16까지, 실험하는걸 추천.

만약 퍼포먼스에서 어떠한 차이도 볼 수 없다면, 이 옵션을 건드리는걸 원하지 않았거나, direct io의 값이 1일 때 차이가 없음

모든 서버와 OS가 다르기 때문에 결과들을 저장하기 전에 여러 조합을 실험해봐라

direct

direct io를 쓰거나 buffered io 쓰는걸 말한다.

default value = 0 은 fio가 buffered io를 사용하는걸 뜻함

만약 값을 1로 정하고 buffered io를 사용하는걸 피한다면, 주 O\_DIRECT와 유사하다. ( 캐쉬를 거치지않는다?)

buffered io값을 사용하는게 non\_buffered io 사용하는 것보다 더 나은 퍼포먼스를 제공한다, 특히 read에서.

만약 ram이 용량이 엄청 큰 서버에서 테스트한다면 non\_buffered io를 쓰는게 부풀어진 결과값을 피하는데 도움이 된다. ssd가 600000 iops값을 계속 수행했다면, 이상한게 아니고, ram을 읽어낸 것, 명확히 엄청빠름

fsync

이 옵션은 얼마나 종종 fsync를 사용하여 disk의 "dirty data"를 씻어낼껀지를 말한다.

by default, fsync=0 은 sync를 하지않는다는걸 뜻한다.

많은 어플리케이션들이 이 옵션으로 수행하고 리눅스에서, 메모리에서 디스크로 데이터를 flush 하는걸 계산할 때 떠난다. 어플리케이션이나 서버가 디스크에 write할 때 항상 flush 한다면 이 옵션을 포함시키고 값을 1로 줘야한다.

어플리케이션이 각각의 write 후에 디스크에서 data를 flush하지 못하거나 잠재적인 데이터손실에 걱정이 없다면 이 값을 alone하게 둬라 (leave this value alone)

fsync값을 1로 하면 완벽하게 읽기의 버퍼링을 피할 수 있다.

만약 block device 에서 io performance의 최악의 경우를 보고 싶다면 fsync를 1로 하고 random write test를 돌리면 결과는 without fsync 보다 낮을 것이지만, single write operation이 디스크로 flush되고, 디스크는 스트레스를 받을 것이다.

--dirctory를 통해 경로를 지정

--name을 통해 이름을 지정

--rw를 통해 read/write를 지정

--bs를 통해 블록 사이즈 지정

--randrepeat

--size 작업당 i/o의 크기 지정

--direct direct i/o / buffered i/o 사용여부 지정

--ioengine 사용할 ioengine을 지정

--iodepth

--numjobs

--time\_based

--runtime

--group\_reporting

### **1.7.2 XFS Options**

# **2. 실험**

## **2.1 Fio benchmark를 통한 Ext4와 XFS 파일시스템 성능비교**

fio명령어의 기본적인 명령인 --dirrectory, --name, --rw, --bs, --size, --runtime, --timebased을 사용했고 size=1G, runtime=3600을 기준으로 bs값에 변화를 주어 실험을 진행했다.

ioengine=psync, iodepth=1로 default값으로 자동으로 진행되었다.

### **2.1.1 Ext4파일시스템에서 bs 변화에 따른 IOPS값과 bw값**

### **2.1.2 XFS 파일 시스템에서 bs 변화에 따른 bw값과 IOPS값**

### **2.1.3 실험결과: 두 파일시스템에서 IOPS값과 bw값 비교**

Ext4에서는 IOPS와 bw 모두 read, randread, write, randwrite 순으로 값의 차이가 있었고,

bs가 증가할 때 IOPS값은 감소했고 bw값은 증가했다.

반면에, XFS에서는 Ext4에 비해 bs가 커짐에 따라 IOPS값이 감소할 때 write와 randwrite가 Ext4보다 감소폭이 적었고, bw값도 wite와 randwrite가 증가 폭이 더 컸다.

Ext4와 XFS의 iops, bs 비교표

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | iops | 4k | 8k | 16k | 32k | 64k |
| read |  | 1.024 | 1.019 | 1.022 | 0.996 | 1 |
| randread |  | 0.98 | 1.017 | 1.01 | 1.008 | 0.993 |
| write |  | 0.963 | 0.806 | 0.725 | 0.687 | 0.606 |
| randwrite |  | 0.988 | 0.859 | 0.782 | 0.694 | 0.698 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | bw | 4k | 8k | 16k | 32k | 64k |
| read |  | 1.025 | 1.066 | 1.024 | 0.996 | 0.997 |
| randread |  | 0.981 | 1.017 | 1.011 | 1.006 | 0.993 |
| write |  | 0.962 | 0.73 | 0.725 | 0.686 | 0.606 |
| randwrite |  | 0.987 | 0.859 | 0.781 | 0.694 | 0.701 |

Ext4의 값/XFS의 값을 구한 것으로 read와 randread는 크게 차이가 없지만

표에서 왼쪽에서 오른쪽으로, 4k~64k 순서로 갈수록 write와 randwrite는

XFS의 값이 더 성능이 좋은 것을 확인 할 수 있다.

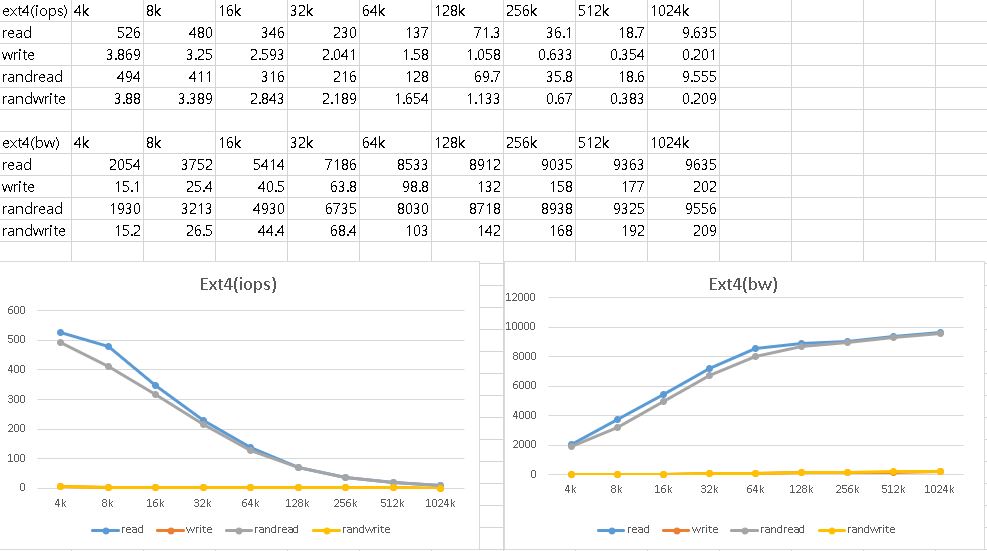
단, randwrite는 32k에서 64k로 갈 때 약간의 값이 증가하는 것을 보아 bs가 커질수록 XFS의 성능이 감소할 수 있을 것 같다.

## **2.2 fsync 옵션을 통한 파일시스템간 성능 비교**

이번 실험에서는 fsync=8 옵션을 추가하고 ioengine=libaio를 사용하였다.

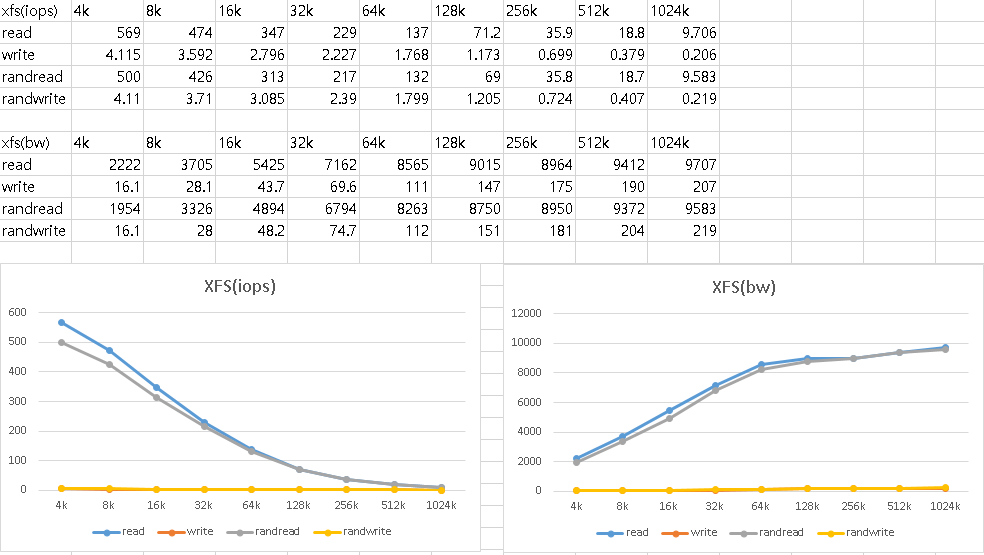
### **2.2.1 두 파일 시스템에서의 결과값**

**Ext4에서의 결과값**



read 와 randread가 write와 randwrite보다 전반적으로 성능이 뛰어난 것을 확인할 수 있다.

**XFS에서의 결과값**



Ext4와 마찬가지로 read, randread가 write, randwrite 보다 성능이 뛰어난 것을 확인할 수 있다.

### **2.2.2 두 파일시스템의 결과값 비교**

Read(sequential read)에서는 ext4와 xfs의 결과값에 큰 차이가 없다.

Xfs가 약간 우세하다.

Write(sequential write)에서도 ext4와 xfs의 결과값에 큰 차이가 없다.

역시 xfs가 약간 우세하다.

Random read에서도 ext4와 xfs의 결과값에 큰 차이가 없다.

xfs가 약간 우세하다.

Random write에서도 ext4와 xfs의 결과값에 큰 차이가 없다.

xfs가 약간 우세하다.

### **2.2.3 실험결과**

Fsync=8 옵션을 부여하고 실험을 한 결과 ext4와 xfs 두 파일시스템에서 현저한 속도의 차이는 발견되지 않았다.

하지만, 옵션을 부여하지 않고 실험을 했을 때는 read, randread와 write, randwrite의 속도 차이가 계단형으로 고르게 속도 차이가 났다면, 옵션을 부여했을 때는 ext4의 경우에는 bw와 iops값이 최대 135.9배에서 최소 45.7배의 속도 차이가 났고, xfs의 경우에는 bw와 iops의 값이 최대 138.2배에서 최소 43.7배의 차이가 났다.

Fsync=8 옵션을 부여한 결과 두 파일시스템간의 큰 속도 차이는 없었고, read부분과 write부분의 속도차이가 현저하게 난 것을 확인할 수 있었다.